ĐẠI HỌC ĐÀ NẴNG
TRƯỜNG ĐẠI HỌC KỸ THUẬT
KHOA CÔNG NGHỆ THÔNG TIN - ĐIỆN TỬ VIỄN THÔNG

GIÁO TRÌNH MÔN HỌC

LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

BIÊN SOẠN: LÊ THỊ MỸ HẠNH

ĐÀ NẴNG, 09/2002

MỤC LỤC

CHƯƠNG 1: GIỚI THIỆU VỀ LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG 5
I. LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG (OOP) LÀ GÌ ? 5
I.1. Lập trình tuyến tính 5
I.2. Lập trình cấu trúc 5
I.3. Sự trừu tượng hóa dữ liệu 6
I.4. Lập trình hướng đối tượng 6
II. MỘT SỐ KHÁI NIỆM MỚI TRONG LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG 8
II.1. Sự đóng gói (Encapsulation) 8
II.2. Tính kế thừa (Inheritance) 9
II.3. Tính đa hình (Polymorphism) 10
III. CÁC NGÔN NGỮ VÀ VÀI ỨNG DỤNG CỦA OOP 11
CHƯƠNG 2: CÁC MỞ RỘNG CỦA C++ 12
I. LỊCH SỬ CỦA C++ 12
II. CÁC MỞ RỘNG CỦA C++ 12
II.1. Các từ khóa mới của C++ 12
II.2. Cách ghi chú thích 12
II.3. Dòng nhập/xuất chuẩn 13
II.4. Cách chuyển đổi kiểu dữ liệu 14
II.5. Vị trí khai báo biến 14
II.6. Các biến const 15
II.7. Về struct, union và enum 16
II.8. Toán tử định phạm vi 16
II.9. Toán tử new và delete 17
II.10. Hàm inline 23
II.11. Các giá trị tham số mặc định 24
II.12. Phép tham chiếu 25
II.13. Phép đa năng hóa (Overloading) 29
CHƯƠNG 3: LỚP VÀ ĐỐI TƯỢNG 39
I. DẪN NHẬP 39
II. CÀI ĐẶT MỘT KIỂU DO NGƯỜI DÙNG ĐỊNH NGHĨA VỚI MỘT STRUCT 39
III. CÀI ĐẶT MỘT KIỂU DỮ LIỆU TRỪU TƯỢNG VỚI MỘT LỚP 41
IV. PHẠM VI LỚP VÀ TRUY CẬP CÁC THÀNH VIÊN LỚP 45
V. ĐIỀU KHIỂN TRUY CẬP TỚI CÁC THÀNH VIÊN 47
VI. CÁC HÀM TRUY CẬP VÀ CÁC HÀM TIỆN ÍCH 48
VII. KHỞI ĐỘNG CÁC ĐỐI TƯỢNG CỦA LỚP : CONSTRUCTOR 49
VIII. SỬ DỤNG DESTRUCTOR 51
IX. KHI NÀO CÁC CONSTRUTOR VÀ DESTRUCTOR ĐƯỢC GỌI ? 53
X. SỬ DỤNG CÁC THÀNH VIÊN DỮ LIỆU VÀ CÁC HÀM THÀNH VIÊN 54
XI. TRẢ VỀ MỘT THAM CHIẾU TỚI MỘT THÀNH VIÊN DỮ LIỆU PRIVATE 57
XII. PHÉP GÁN BỞI TOÁN TỬ SAO CHÉP THÀNH VIÊN MẶC ĐỊNH 59
XIII. CÁC ĐỐI TƯỢNG HẰNG VÀ CÁC HÀMTHÀNH VIÊN CONST 60
XIV. LỚP NHƯ LÀ CÁC THÀNH VIÊN CỦA CÁC LỚP KHÁC 64
XV. CÁC HÀM VÀ CÁC LỚP FRIEND 67

XVI.CON TRỎ THIS 68
XVII.CÁC ĐỐI TƯỢNG ĐƯỢC CẤP PHÁT ĐỘNG 71
XVIII.CÁC THÀNH VIÊN TĨNH CỦA LỚP 72

CHƯƠNG 4: ĐA NĂNG HÓA TOÁN TỬ 76
I. DẪN NHẬP 76
II. CÁC NGUYÊN TẮC CƠ BẢN CỦA ĐA NĂNG HÓA TOÁN TỬ 76
III. CÁC GIỚI HẠN CỦA ĐA NĂNG HÓA TOÁN TỬ 76
IV. CÁC HÀM TOÁN TỬ CÓ THỂ LÀ CÁC THÀNH VIÊN CỦA LỚP HOẶC KHÔNG LÀ CÁC THÀNH VIÊN 77
V. ĐA NĂNG HOÁ CÁC TOÁN TỬ HAI NGÔI 80
VI. ĐA NĂNG HÓA CÁC TOÁN TỬ MỘT NGÔI 87
VII. ĐA NĂNG HÓA MỘT SỐ TOÁN TỬ ĐẶC BIỆT 90
VII.1.Toán tử [] 91
VII.2.Toán tử () 92
VIII.TOÁN TỬ CHUYỂN ĐỔI KIỂU 94
IX. TOÁN TỬ NEW VÀ DELETE 95
IX.1.Đa năng hóa toán tử new và delete toàn cục 96
IX.2.Đa năng hóa toán tử new và delete cho một lớp 97
X. ĐA NĂNG HÓA CÁC TOÁN TỬ CHÈN DÒNG << VÀ TRÍCH DÒNG >> 98
XI. MỘT SỐ VÍ DỤ 99
XI.1.Lớp String 99
XI.2.Lớp Date 103

CHƯƠNG 5: TÍNH KẾ THỪA 107
I. DẪN NHẬP 107
II. KẾ THỪA ĐƠN 107
II.1.Các lớp cơ sở và các lớp dẫn xuất 107
II.2.Các thành viên protected 109
II.3.Ép kiểu các con trỏ lớp cơ sở tới các con trỏ lớp dẫn xuất 109
II.4.Định nghĩa lại các thành viên lớp cơ sở trong một lớp dẫn xuất: 113
II.5.Các lớp cơ sở public, protected và private 113
II.6.Các contructor và destructor lớp dẫn xuất 113
II.7.Chuyển đổi ngầm định đối tượng lớp dẫn xuất sang đối tượng lớp cơ sở 116
III. ĐA KẾ THỪA (MULTIPLE INHERITANCE) 116
IV. CÁC LỚP CƠ SỞ ẢO (VIRTUAL BASE CLASSES) 119

CHƯƠNG 6: TÍNH ĐA HÌNH 122
I. DẪN NHẬP 122
II. PHƯƠNG THỨC ẢO (VIRTUAL FUNCTION) 122
III. LỚP TRỪU TƯỢNG (ABSTRACT CLASS) 125
IV. CÁC THÀNH VIÊN ẢO CỦA MỘT LỚP 127
IV.1.Toán tử ảo 127
IV.2.Có constructor và destructor ảo hay không? 129

CHƯƠNG 7: THIẾT KẾ CHƯƠNG TRÌNH THEO HƯỚNG ĐỐI TƯỢNG 132
I. DẪN NHẬP 132
II. CÁC GIAI ĐOẠN PHÁT TRIỂN HỆ THỐNG 132
III. CÁCH TÌM LỚP 133
IV. CÁC BƯỚC CẦN THIẾT ĐỂ THIẾT KẾ CHƯƠNG TRÌNH 133
V. CÁC VÍ DỤ 134

CHƯƠNG 8: CÁC DẠNG NHẬP/XUẤT 143
I. DẪN NHẬP 143
II. CÁC DÒNG(STREAMS) 143
II.1.Các file header của thư viện iostream 143
II.2.Các lớp và các đối tượng của dòng nhập/xuất 144
III. DÒNG XUẤT 145
III.1.Toán tử chèn dòng 145
III.2.Nối các toán tử chèn dòng và trích dòng 146
III.3.Xuất ký tự với hàm thành viên put(); Nối với nhau hàm put() 147
IV. DÒNG NHẬP 148
IV.1.Toán tử trích dòng: 148
IV.2.Các hàm thành viên get() và getline() 149
IV.3.Các hàm thành viên khác của istream 151
IV.4.Nhập/xuất kiểu an toàn 151
V. NHẬP/XUẤT KHÔNG ĐỊNH DẠNG VỚI READ(),GCOUNT() VÀ WRITE() 151
VI. DÒNG NHẬP/ XUẤT FILE 152
VI.1.Nhập/xuất file văn bản 154

CHƯƠNG 9: HÀM VÀ LỚP TEMPLATE 159
I. CÁC HÀM TEMPLATE 159
II. CÁC LớP TEMPLATE 161

CHƯƠNG 1

GIỚI THIỆU VỀ LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

I. LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG (OOP) LÀ GÌ ?

Lập trình hướng đối tượng (Object-Oriented Programming, viết tắt là OOP) là một phương pháp mới trên bước đường tiến hóa của việc lập trình máy tính, nhằm làm cho chương trình trở nên linh hoạt, tin cậy và dễ phát triển. Tuy nhiên để hiểu được OOP là gì, chúng ta hãy bắt đầu từ lịch sử của quá trình lập trình – xem xét OOP đã tiến hóa như thế nào.

I.1. Lập trình tuyến tính

Máy tính đầu tiên được lập trình bằng mã nhị phân, sử dụng các công tắt cơ khí để nạp chương trình. Cùng với sự xuất hiện của các thiết bị lưu trữ lớn và bộ nhớ máy tính có dung lượng lớn nên các ngôn ngữ lập trình cấp cao đầu tiên được đưa vào sử dụng . Thay vì phải suy nghĩ trên một dãy các bit và byte, lập trình viên có thể viết một loạt lệnh gần với tiếng Anh và sau đó chương trình dịch thành ngôn ngữ máy.

Các ngôn ngữ lập trình cấp cao đầu tiên được thiết kế để lập các chương trình làm các công việc tương đối đơn giản như tính toán. Các chương trình ban đầu chủ yếu liên quan đến tính toán và không đòi hỏi gì nhiều ở ngôn ngữ lập trình. Hơn nữa phần lớn các chương trình này tương đối ngắn, thường ít hơn 100 dòng.

Khi khả năng của máy tính tăng lên thì khả năng để triển khai các chương trình phức tạp hơn cũng tăng lên. Các ngôn ngữ lập trình ngày trước không còn thích hợp đối với việc lập trình đòi hỏi cao hơn. Các phương tiện cần thiết để sử dụng lại các phần mã chương trình đã viết hầu như không có trong ngôn ngữ lập trình tuyến tính. Thật ra, một đoạn lệnh thường phải được chép lặp lại mỗi khi chúng ta dùng trong nhiều chương trình do đó chương trình dài dòng, logic của chương trình khó hiểu. Chương trình được điều khiển để nhảy đến nhiều chỗ mà thường không có sự giải thích rõ ràng, làm thế nào để chương trình đến chỗ cần thiết hoặc tại sao như vậy.

Ngôn ngữ lập trình tuyến tính không có khả năng kiểm soát phạm vi nhìn thấy của các dữ liệu. Mọi dữ liệu trong chương trình đều là dữ liệu toàn cục nghĩa là chúng có thể bị sửa đổi ở bất kỳ phần nào của chương trình. Việc dò tìm các thay đổi không mong muốn đó của các phần tử dữ liệu trong một dãy mã lệnh dài và vòng vèo đã từng làm cho các lập trình viên rất mất thời gian.

I.2. Lập trình cấu trúc

Rõ ràng là các ngôn ngữ mới với các tính năng mới cần phải được phát triển để có thể tạo ra các ứng dụng tinh vi hơn. Vào cuối các năm trong 1960 và 1970, ngôn ngữ lập trình có cấu trúc ra đời. Các chương trình có cấu trúc được tổ chức theo các công việc mà chúng thực hiện.

Về bản chất, chương trình chia nhỏ thành các chương trình con riêng rẽ (còn gọi là hàm hay thủ tục) thực hiện các công việc rời rạc trong quá trình lớn hơn, phức tạp hơn. Các hàm này được giữ càng độc lập với nhau càng nhiều càng tốt, mỗi hàm có dữ liệu và logic riêng.Thông tin được chuyển giao giữa các hàm thông qua các tham số, các hàm có thể có các biến cục bộ mà không một ai nằm bên ngoài phạm vi của hàm lại có thể truy xuất được chúng. Như vậy, các hàm có thể được xem là các chương trình con được đặt chung với nhau để xây dựng nên một ứng dụng.

Mục tiêu là làm sao cho việc triển khai các phần mềm dễ dàng hơn đối với các lập trình viên mà vẫn cải thiện được tính tin cậy và dễ bảo quản chương trình. Một chương trình có cấu trúc được hình thành bằng cách bẻ gãy các chức năng cơ bản của chương trình thành các mảnh nhỏ mà sau đó trở thành các hàm. Bằng cách cô lập các công việc vào trong các hàm, chương trình có cấu trúc có thể làm giảm khả năng của một hàm này ảnh hưởng đến một hàm khác. Việc này cũng làm cho việc tách các vấn đề trở nên dễ dàng hơn. Sự gói gọn này cho phép chúng ta có thể viết các chương trình sáng sủa hơn và giữ được điều khiển trên từng hàm. Các biến toàn cục không còn nữa và được thay thế bằng các tham số và biến cục bộ có phạm vi nhỏ hơn và dễ kiểm soát hơn. Cách tổ chức tốt hơn này nói lên rằng chúng ta có khả năng quản lý logic của cấu trúc chương trình, làm cho việc triển khai và bảo dưỡng chương trình nhanh hơn và hữu hiện hơn và hiệu quả hơn.

Một khái niệm lớn đã được đưa ra trong lập trình có cấu trúc là **sự trừu tượng hóa** (**Abstraction**). Sự trừu tượng hóa có thể xem như khả năng quan sát một sự việc mà không cần xem xét đến các chi tiết bên trong của nó. Trong một chương trình có cấu trúc, chúng ta chỉ cần biết một hàm đã cho có thể làm được một công việc cụ thể gì là đủ. Còn làm thế nào mà công việc đó lại thực hiện được là không quan trọng, chừng nào hàm còn tin cậy được thì còn có thể dùng nó mà không cần phải biết nó thực hiện đúng đắn chức năng của mình như thế nào. Điều này gọi là **sự trừu tượng hóa theo chức năng** (**Functional abstraction**) và là nền tảng của lập trình có cấu trúc.

Ngày nay, các kỹ thuật thiết kế và lập trình có cấu trúc được sử rộng rãi. Gần như mọi ngôn ngữ lập trình đều có các phương tiện cần thiết để cho phép lập trình có cấu trúc. Chương trình có cấu trúc dễ viết, dễ bảo dưỡng hơn các chương trình không cấu trúc.

Sự nâng cấp như vậy cho các kiểu dữ liệu trong các ứng dụng mà các lập trình viên đang viết cũng đang tiếp tục diễn ra. Khi độ phức tạp của một chương trình tăng lên, sự phụ thuộc của nó vào các kiểu dữ liệu cơ bản mà nó xử lý cũng tăng theo. Vấn đề trở rõ ràng là cấu trúc dữ liệu trong chương trình quan trọng chẳng kém gì các phép toán thực hiện trên chúng. Điều này càng trở rõ ràng hơn khi kích thước của chương trình càng tăng. Các kiểu dữ liệu được xử lý trong nhiều hàm khác nhau bên trong một chương trình có cấu trúc. Khi có sự thay đổi trong các dữ liệu này thì cũng cần phải thực hiện cả các thay đổi ở mọi nơi có các thao tác tác động trên chúng. Đây có thể là một công việc tốn thời gian và kém hiệu quả đối với các chương trình có hàng ngàn dòng lệnh và hàng trăm hàm trở lên.

Một yếu điểm nữa của việc lập trình có cấu trúc là khi có nhiều lập trình viên làm việc theo nhóm cùng một ứng dụng nào đó. Trong một chương trình có cấu trúc, các lập trình viên được phân công viết một tập hợp các hàm và các kiểu dữ liệu. Vì có nhiều lập trình viên khác nhau quản lý các hàm riêng, có liên quan đến các kiểu dữ liệu dùng chung nên các thay đổi mà lập trình viên tạo ra trên một phần tử dữ liệu sẽ làm ảnh hưởng đến công việc của tất cả các người còn lại trong nhóm. Mặc dù trong bối cảnh làm việc theo nhóm, việc viết các chương trình có cấu trúc thì dễ dàng hơn nhưng sai sót trong việc trao đổi thông tin giữa các thành viên trong nhóm có thể dẫn tới hậu quả là mất rất nhiều thời gian để sửa chữa chương trình.

I.3. Sự trừu tượng hóa dữ liệu

Sự trừu tượng hóa dữ liệu (**Data abstraction**) tác động trên các dữ liệu cũng tương tự như sự trừu tượng hóa theo chức năng. Khi có trừu tượng hóa dữ liệu, các cấu trúc dữ liệu và các phần tử có thể được sử dụng mà không cần bận tâm đến các chi tiết cụ thể. Chẳng hạn như các số dấu chấm động đã được trừu tượng hóa trong tất cả các ngôn ngữ lập trình, Chúng ta không cần quan tâm cách biểu diễn nhị phân chính xác nào cho số dấu chấm động khi gán một giá trị, cũng không cần biết tính bất thường của phép nhân nhị phân khi nhân các giá trị dấu chấm động. Điều quan trọng là các số dấu chấm động hoạt động đúng đắn và hiểu được.

Sự trừu tượng hóa dữ liệu giúp chúng ta không phải bận tâm về các chi tiết không cần thiết. Nếu lập trình viên phải hiểu biết về tất cả các khía cạnh của vấn đề, ở mọi lúc và về tất cả các hàm của chương trình thì chỉ ít hàm mới được viết ra, may mắn thay trừu tượng hóa theo dữ liệu đã tồn tại sẵn trong mọi ngôn ngữ lập trình đối với các dữ liệu phức tạp như số dấu chấm động. Tuy nhiên chỉ mới gần đây, người ta mới phát triển các ngôn ngữ cho phép chúng ta định nghĩa các kiểu dữ liệu trừu tượng riêng.

I.4. Lập trình hướng đối tượng

Khái niệm hướng đối tượng được xây dựng trên nền tảng của khái niệm lập trình có cấu trúc và sự trừu tượng hóa dữ liệu. Sự thay đổi căn bản ở chỗ, một chương trình hướng đối tượng được thiết kế xoay quanh dữ liệu mà chúng ta có thể làm việc trên đó, hơn là theo bản thân chức năng của chương trình. Điều này hoàn toàn tự nhiên một khi chúng ta hiểu rằng mục tiêu của chương trình là xử lý dữ liệu. Suy cho cùng, công việc mà máy tính thực hiện vẫn thường được gọi là xử lý dữ liệu. Dữ liệu và thao tác liên kết với nhau ở một mức cơ bản (còn có thể gọi là mức thấp), mỗi thứ đều đòi hỏi ở thứ kia có mục tiêu cụ thể, các chương trình hướng đối tượng làm tường minh mối quan hệ này.

Lập trình hướng đối tượng (Object Oriented Programming - gọi tắt là OOP) hay chi tiết hơn là *Lập trình định hướng đối tượng,* chính là phương pháp lập trình lấy đối tượng làm nền tảng để xây dựng thuật giải, xây

dựng chương trình. Thực chất đây không phải là một phương pháp mới mà là một cách nhìn mới trong việc lập trình. Để phân biệt, với phương pháp lập trình theo kiểu cấu trúc mà chúng ta quen thuộc trước đây, hay còn gọi là phương pháp lập trình hướng thủ tục (Procedure-Oriented Programming), người lập trình phân tích một nhiệm vụ lớn thành nhiều công việc nhỏ hơn, sau đó dần dần chi tiết, cụ thể hoá để được các vấn đề đơn giản, để tìm ra cách giải quyết vấn đề dưới dạng những thuật giải cụ thể rõ ràng qua đó dễ dàng minh hoạ bằng ngôn ngữ giải thuật (hay còn gọi các thuật giải này là các chương trình con). Cách thức phân tích và thiết kế như vậy chúng ta gọi là nguyên lý lập trình từ trên xuống (*top-down)*, để thể hiện quá trình suy diễn từ cái chung cho đến cái cụ thể.

Các chương trình con là những chức năng độc lập, sự ghép nối chúng lại với nhau cho chúng ta một hệ thống chương trình để giải quyết vấn đề đặt ra. Chính vì vậy, cách thức phân tích một hệ thống lấy chương trình con làm nền tảng, chương trình con đóng vai trò trung tâm của việc lập trình, được hiểu như phương pháp lập trình hướng về thủ tục. Tuy nhiên, khi phân tích để thiết kế một hệ thống không nhất thiết phải luôn luôn suy nghĩ theo hướng "*làm thế nào để giải quyết công việc"*, chúng ta có thể định hướng tư duy theo phong cách "*với một số đối tượng đã có, phải làm gì để giải quyết được công việc đặt ra*" hoặc phong phú hơn, "*làm cái gì với một số đối tượng đã có đó*", từ đó cũng có thể giải quyết được những công việc cụ thể. Với phương pháp phân tích trong đó đối tượng đóng vai trò trùng tâm của việc lập trình như vậy, người ta gọi là nguyên lý lập trình từ dưới lên (*Bôttm-up)*.

Lập trình hướng đối tượng liên kết cấu trúc dữ liệu với các thao tác, theo cách mà tất cả thường nghĩ về thế giới quanh mình. Chúng ta thường gắn một số các hoạt động cụ thể với một loại hoạt động nào đó và đặt các giả thiết của mình trên các quan hệ đó.

Ví dụ1.1: Để dễ hình dùng hơn, chúng ta thủ nhìn qua các công trình xây dựng hiện đại, như sân vận động có mái che hình vòng cung, những kiến trúc thẩm mĩ với đường nét hình cong. Tất cả những sản phẩm đó xuất hiện cùng với những vật liệu xây dựng. Ngày nay, không chỉ chồng lên nhau những viên gạch, những tảng đá để tạo nên những quần thể kiến trúc (như Tháp Chàm Nha Trang, Kim Tự Tháp,...), mà có thể với bêtông, sắt thép và không nhiều lắm những viên gạch, người xây dựng cũng có thể thiết kế những công trình kiến trúc tuyệt mỹ, những toà nhà hiện đại. Chính các chất liệu xây dựng đã làm ảnh hưởng phương pháp xây dựng, chất liệu xây dựng và nguyên lý kết dính caá chất liệu đó lại với nhau cho chúng ta một đối tượng để khảo sát, Chất liệu xây dựng và nguyên lý kết dính các chất liệu lại với nhau được hiểu theo nghĩa dữ liệu và chương trình con tác động trên dữ liệu đó.

Ví dụ1.2: Chúng ta biết rằng một chiếc xe có các bánh xe, di chuyển được và có thể đổi hướng của nó bằng cách quẹo tay lái. Tương tự như thế, một cái cây là một loại thực vật có thân gỗ và lá. Một chiếc xe không phải là một cái cây, mà cái cây không phải là một chiếc xe, chúng ta có thể giả thiết rằng cái mà chúng ta có thể làm được với một chiếc xe thì không thể làm được với một cái cây. Chẳng hạn, thật là vô nghĩa khi muốn lái một cái cây, còn chiếc xe thì lại chẳng lớn thêm được khi chúng ta tưới nước cho nó.

Lập trình hướng đối tượng cho phép chúng ta sử dụng các quá trình suy nghĩ như vậy với các khái niệm trừu tượng được sử dụng trong các chương trình máy tính. Một mẫu tin (record) nhân sự có thể được đọc ra, thay đổi và lưu trữ lại; còn số phức thì có thể được dùng trong các tính toán. Tuy vậy không thể nào lại viết một số phức vào tập tin làm mẫu tin nhân sự và ngược lại hai mẫu tin nhân sự lại không thể cộng với nhau được. Một chương trình hướng đối tượng sẽ xác định đặc điểm và hành vi cụ thể của các kiểu dữ liệu, điều đó cho phép chúng ta biết một cách chính xác rằng chúng ta có thể có được những gì ở các kiểu dữ liệu khác nhau.

Chúng ta còn có thể tạo ra các quan hệ giữa các kiểu dữ liệu tương tự nhưng khác nhau trong một chương trình hướng đối tượng. Người ta thường tự nhiên phân loại ra mọi thứ, thường đặt mối liên hệ giữa các khái niệm mới với các khái niệm đã có, và thường có thể thực hiện suy diễn giữa chúng trên các quan hệ đó. Hãy quan niệm thế giới theo kiểu cấu trúc cây, với các mức xây dựng chi tiết hơn kế tiếp nhau cho các thế hệ sau so với các thế hệ trước. Đây là phương pháp hiệu quả để tổ chức thế giới quanh chúng ta. Các chương trình hướng đối tượng cũng làm việc theo một phương thức tương tự, trong đó chúng cho phép xây dựng các các cơ cấu dữ liệu và thao tác mới dựa trên các cơ cấu có sẵn, mang theo các tính năng của các cơ cấu nền mà chúng dựa trên đó, trong khi vẫn thêm vào các tính năng mới.

Lập trình hướng đối tượng cho phép chúng ta tổ chức dữ liệu trong chương trình theo một cách tương tự như các nhà sinh học tổ chức các loại thực vật khác nhau. Theo cách nói lập trình đối tượng, xe hơi, cây cối, các số phức, các quyển sách đều được gọi là các **lớp** (**Class**).

Một lớp là một bản mẫu mô tả các thông tin cấu trúc dữ liệu, lẫn các thao tác hợp lệ của các phần tử dữ liệu. Khi một phần tử dữ liệu được khai báo là phần tử của một lớp thì nó được gọi là một **đối tượng** (**Object**). Các hàm được định nghĩa hợp lệ trong một lớp được gọi là các **phương thức** (**Method**) và chúng là các hàm duy nhất có thể xử lý dữ liệu của các đối tượng của lớp đó. Một **thực thể** (**Instance**) là một vật thể có thực bên trong bộ nhớ, thực chất đó là một đối tượng (nghĩa là một đối tượng được cấp phát vùng nhớ).

Mỗi một đối tượng có riêng cho mình một bản sao các phần tử dữ liệu của lớp còn gọi là các **biến thực thể** (**Instance variable**). Các phương thức định nghĩa trong một lớp có thể được gọi bởi các đối tượng của lớp đó. Điều này được gọi là gửi một **thông điệp** (**Message**) cho đối tượng. Các thông điệp này phụ thuộc vào đối tượng, chỉ đối tượng nào nhận thông điệp mới phải làm việc theo thông điệp đó. Các đối tượng đều độc lập với nhau vì vậy các thay đổi trên các biến thể hiện của đối tượng này không ảnh hưởng gì trên các biến thể hiện của các đối tượng khác và việc gửi thông điệp cho một đối tượng này không ảnh hưởng gì đến các đối tượng khác.

Như vậy, đối tợng được hiểu theo nghĩa là một thực thể mà trong đó caá dữ liệu và thủ tục tác động lên dữ liệu đã được đóng gói lại với nhau. Hay "*đối tượng được đặc trưng bởi một số thao tác (operation) và các thông tin (information) ghi nhơ sự tác động của caá thao tác này.*"

Ví dụ 1.3: Khi nghiên cứ về ngăn xếp (*stack)*, ngoài các dữ liệu vùng chứa ngăn xếp, đỉnh của ngăn xếp, chúng ta phải cài đặt kèm theo các thao tác như khởi tạo (*creat*) ngăn xếp, kiểm tra ngăn xếp rỗng (*empty*), đẩy (*push*) một phần tử vào ngăn xếp, lấy (*pop*) một phần tử ra khỏi ngăn xếp. Trên quan điểm lấy đối tượng làm nền tảng, rõ ràng dữ liệu và các thao tác trên dữ liệu luôn gắn bó với nhau, sự kết dính chúng chính là đối tượng chúng ta cần khảo sát.

Các thao tác trong đối tượng được gọi là các phương thức hay hành vi của đối tượng đó. Phương thức và dữ liệu của đối tượng luôn tác động lẫn nhau và có vai trò ngang nhau trong đối tượng, Phương thức của đối tượng được qui định bởi dữ liệu và ngược lại, dữ liệu của đối tượng được đặt trưng bởi các phương thức của đối tượng. Chính nhờ sự gắn bó đó, chúng ta có thể gởi cùng một thông điệp đến những đối tượng khác nhau. Điều này giúp người lập trình không phải xử lý trong chương trình của mình một dãy các cấu trúc điều khiển tuỳ theo thông điệp nhận vào, mà chương trình được xử lý vào thời điểm thực hiện.

Tóm lại, so sánh lập trình cấu trúc với chương trình con làm nền tảng:

Chương trình = Cấu trúc dữ liệu + Thuật giải

Trong lập trình hướng đối tượng chúng ta có:

Đối tượng = Phương thức + Dữ liệu

Đây chính là 2 quan điểm lập trình đang tồn tại và phát triển trong thế giới ngày nay.

II. MỘT SỐ KHÁI NIỆM MỚI TRONG LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

Trong phần này, chúng ta tìm hiểu các khái niệm như sự đóng gói, tính kế thừa và tính đa hình. Đây là các khái niệm căn bản, là nền tảng tư tưởng của lập trình hướng đối tượng. Hiểu được khái niệm này, chúng ta bước đầu tiếp cận với phong cách lập trình mới, phong cách lập trình dựa vào đối tượng làm nền tảng mà trong đó quan điểm che dấu thông tin thông qua sư đóng gói là quan điểm trung tâm của vấn đề.

II.1. Sự đóng gói (Encapsulation)

Sự đóng gói là cơ chế ràng buộc dữ liệu và thao tác trên dữ liệu đó thành một thể thống nhất, tránh được các tác động bất ngờ từ bên ngoài. Thể thống nhất này gọi là đối tượng.

Trong *Objetc Oriented Software Engineering* của *Ivar Jacibson,* tất cả các thông tin của một hệ thống định hướng đối tượng được lưu trữ bên trong đối tượng của nó và chỉ có thể hành động khi các đối tượng đó được ra lệnh thực hiện các thao tác. Như vật, sự đóng gói không chỉ đơn thuần là sự gom chung dữ liệu và chương trình vào trong một khối, chúng còn được hiểu theo nghĩa là sự đồng nhất giữa dữ liệu và các thao tác tác động lên dữ liệu đó.

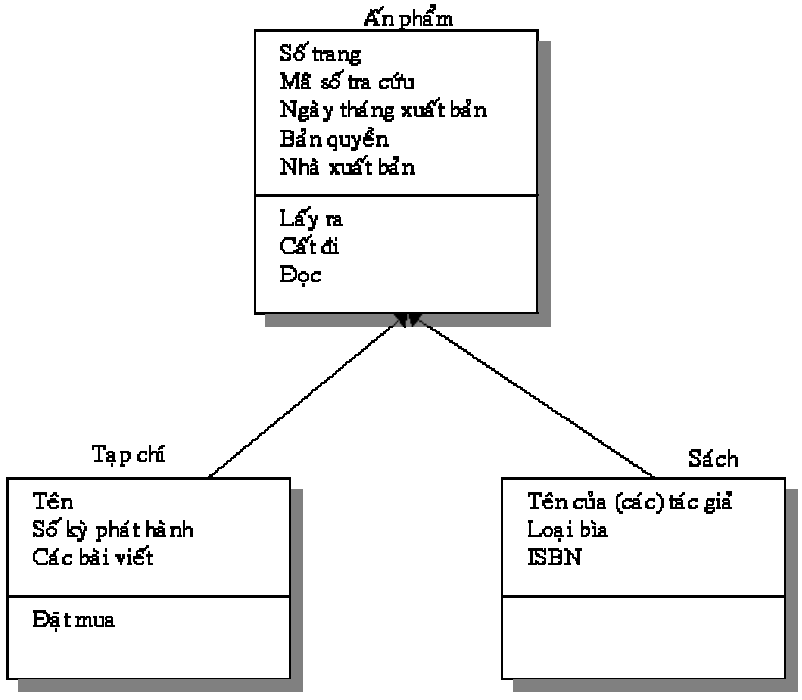
Trong một đối tượng, dữ liệu hay thao tác hay cả hai có thể là **riêng** (**private**) hoặc **chung** (**public**) của đối tượng đó. Thao tác hay dữ liệu riêng là thuộc về đối tượng đó chỉ được truy cập bởi các thành phần của đối tượng, điều này nghĩa là thao tác hay dữ liệu riêng không thể truy cập bởi các phần khác của chương trình tồn tại ngoài đối tượng. Khi thao tác hay dữ liệu là chung, các phần khác của chương trình có thể truy cập nó mặc dù nó được định nghĩa trong một đối tượng. Các thành phần chung của một đối tượng dùng để cung cấp một giao diện có điều khiển cho các thành thành riêng của đối tượng.

Cơ chế đóng gói là phương thức tốt để thực hiện cơ chế che dấu thông tin so với các ngôn ngữ lập trình cấu trúc.

II.2. Tính kế thừa (Inheritance)

Chúng ta có thể xây dựng các lớp mới từ các lớp cũ thông qua sự kế thừa. Một lớp mới còn gọi là **lớp dẫn xuất** (**derived class**), có thể thừa hưởng dữ liệu và các phương thức của **lớp cơ sở** (**base class**) ban đầu. Trong lớp này, có thể bổ sung các thành phần dữ liệu và các phương thức mới vào những thành phần dữ liệu và các phương thức mà nó thừa hưởng từ lớp cơ sở. Mỗi lớp (kể cả lớp dẫn xuất) có thể có một số lượng bất kỳ các lớp dẫn xuất. Qua cơ cấu kế thừa này, dạng hình cây của các lớp được hình thành. Dạng cây của các lớp trông giống như các cây gia phả vì thế các lớp cơ sở còn được gọi là **lớp cha** (**parent class**) và các lớp dẫn xuất được gọi là **lớp con** (**child class**).

Ví dụ 1.2: Chúng ta sẽ xây dựng một tập các lớp mô tả cho thư viện các ấn phẩm. Có hai kiểu ấn phẩm: tạp chí và sách. Chúng ta có thể tạo một ấn phẩm tổng quát bằng cách định nghĩa các thành phần dữ liệu tương ứng với số trang, mã số tra cứu, ngày tháng xuất bản, bản quyền và nhà xuất bản. Các ấn phẩm có thể được lấy ra, cất đi và đọc. Đó là các phương thức thực hiện trên một ấn phẩm. Tiếp đó chúng ta định nghĩa hai lớp dẫn xuất tên là tạp chí và sách. Tạp chí có tên, số ký phát hành và chứa nhiều bài của các tác giả khác nhau . Các thành phần dữ liệu tương ứng với các yếu tố này được đặt vào định nghĩa của lớp tạp chí. Tạp chí cũng cần có một phương thức nữa đó là đặt mua. Các thành phần dữ liệu xác định cho sách sẽ bao gồm tên của (các) tác giả, loại bìa (cứng hay mềm) và số hiệu ISBN của nó. Như vậy chúng ta có thể thấy, sách và tạp chí có chung các đặc trưng ấn phẩm, trong khi vẫn có các thuộc tính riêng của chúng.



Hình 1.1: Lớp ấn phẩm và các lớp dẫn xuất của nó.

Với tính kế thừa, chúng ta không phải mất công xây dựng lại từ đầu các lớp mới, chỉ cần bổ sung để có được trong các lớp dẫn xuất các đặc trưng cần thiết.

II.3. Tính đa hình (Polymorphism)

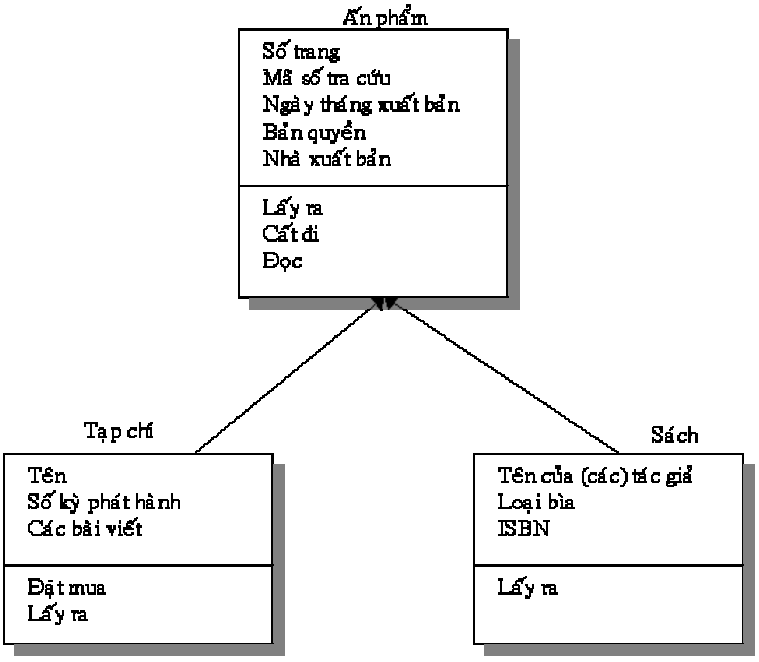
Đó là khả năng để cho một thông điệp có thể thay đổi cách thực hiện của nó theo lớp cụ thể của đối tượng nhận thông điệp. Khi một lớp dẫn xuất được tạo ra, nó có thể thay đổi cách thực hiện các phương thức nào đó mà nó thừa hưởng từ lớp cơ sở của nó. Một thông điệp khi được gởi đến một đối tượng của lớp cơ sở, sẽ dùng phương thức đã định nghĩa cho nó trong lớp cơ sở. Nếu một lớp dẫn xuất định nghĩa lại một phương thức thừa hưởng từ lớp cơ sở của nó thì một thông điệp có cùng tên với phương thức này, khi được gởi tới một đối tượng của lớp dẫn xuất sẽ gọi phương thức đã định nghĩa cho lớp dẫn xuất.

Như vậy đa hình là khả năng cho phép gởi cùng một thông điệp đến những đối tượng khác nhau có cùng chung một đặc điểm, nói cách khác thông điệp được gởi đi không cần biết thực thể nhận thuộc lớp nào, chỉ biết rằng tập hợp các thực thể nhận có chung một tính chất nào đó. Chẳng hạn, thông điệp "*vẽ hình*" được gởi đến cả hai đối tượng hình hộp và hình tròn. Trong hai đối tượng này đều có chung phương thức vẽ hình, tuy nhiên tuỳ theo thời điểm mà đối tượng nhận thông điệp, hình tương ứng sẽ được vẽ lên.

Trong các ngôn ngữ lập trình OOP, tính đa hình thể hiện qua khả năng cho phép mô tả những phương thức có tên giống nhau trong các lớp khác nhau. Đặc điểm này giúp người lập trình không phải viết những cấu trúc điều khiển rườm rà trong chương trình, các khả năng khác nhau của thông điệp chỉ thực sự đòi hỏi khi chương trình thực hiện.

Ví dụ 1.3: Xét lại ví dụ 1.2, chúng ta thấy rằng cả tạp chí và và sách đều phải có khả năng lấy ra. Tuy nhiên phương pháp lấy ra cho tạp chí có khác so với phương pháp lấy ra cho sách, mặc dù kết quả cuối cùng giống nhau. Khi phải lấy ra tạp chí, thì phải sử dụng phương pháp lấy ra riêng cho tạp chí (dựa trên một bản tra cứu) nhưng khi lấy ra sách thì lại phải sử dụng phương pháp lấy ra riêng cho sách (dựa trên hệ thống phiếu lưu trữ). Tính đa hình cho phép chúng ta xác định một phương thức để lấy ra một tạp chí hay một cuốn sách. Khi lấy ra một tạp chí nó sẽ dùng phương thức lấy ra dành riêng cho tạp chí, còn khi lấy ra một cuốn sách thì nó sử dụng phương thức lấy ra tương ứng với sách. Kết quả là chỉ cần một tên phương thức duy nhất được dùng cho cả hai công việc tiến hành trên hai lớp dẫn xuất có liên quan, mặc dù việc thực hiện của phương thức đó thay đổi tùy theo từng lớp.

Tính đa hình dựa trên sự nối kết (Binding), đó là quá trình gắn một phương thức với một hàm thực sự. Khi các phương thức kiểu đa hình được sử dụng thì trình biên dịch chưa thể xác định hàm nào tương ứng với phương thức nào sẽ được gọi. Hàm cụ thể được gọi sẽ tuỳ thuộc vào việc phần tử nhận thông điệp lúc đó là thuộc lớp nào, do đó hàm được gọi chỉ xác định được vào lúc chương trình chạy. Điều này gọi là sự kết nối muộn (Late binding) hay kết nối lúc chạy (Runtime binding) vì nó xảy ra khi chương trình đang thực hiện.



Hình 1.2: Minh họa tính đa hình đối với lớp ấn phẩm và các lớp dẫn xuất của nó.

III. CÁC NGÔN NGỮ VÀ VÀI ỨNG DỤNG CỦA OOP

Xuất phát từ tư tưởng của ngôn ngữ SIMULA67, trung tâm nghiên cứu Palo Alto (PARC) của hãng XEROR đã tập trung 10 năm nghiên cứu để hoàn thiện ngôn ngữ OOP đầu tiên với tên gọi là Smalltalk. Sau đó các ngôn ngữ OOP lần lượt ra đời như Eiffel, Clos, Loops, Flavors, Object Pascal, Object C, C++, Delphi, Java…

Chính XEROR trên cơ sở ngôn ngữ OOP đã đề ra tư tưởng giao diện biểu tượng trên màn hình (icon base screen interface), kể từ đó Apple Macintosh cũng như Microsoft Windows phát triển giao diện đồ họa như ngày nay. Trong Microsoft Windows, tư tưởng OOP được thể hiện một cách rõ nét nhất đó là "chúng ta click vào đối tượng", mỗi đối tượng có thể là control menu, control menu box, menu bar, scroll bar, button, minimize box, maximize box, … sẽ đáp ứng công việc tùy theo đặc tính của đối tượng. Turbo Vision của hãng Borland là một ứng dụng OOP tuyệt vời, giúp lập trình viên không quan tâm đến chi tiết của chương trình gia diện mà chỉ cần thực hiện các nội dung chính của vấn đề.

CHƯƠNG 2

CÁC MỞ RỘNG CỦA C++

I. LỊCH SỬ CỦA C++

Vào những năm đầu thập niên 1980, người dùng biết C++ với tên gọi "C with Classes" được mô tả trong hai bài báo của Bjarne Stroustrup (thuộc AT&T Bell Laboratories) với nhan đề "Classes: An Abstract Data Type Facility for the C Language" và "Adding Classes to C : AnExercise in Language Evolution". Trong công trình này, tác giả đã đề xuất khái niệm lớp, bổ sung việc kiểm tra kiểu tham số của hàm, các chuyển đổi kiểu và một số mở rộng khác vào ngôn ngữ C. Bjarne Stroustrup nghiên cứu mở rộng ngôn ngữ C nhằm đạt đến một ngôn ngữ mô phỏng (simulation language) với những tính năng hướng đối tượng.

Trong năm 1983, 1984, ngôn ngữ "C with Classes" được thiết kế lại, mở rộng hơn rồi một trình biên dịch ra đời. Và chính từ đó, xuất hiện tên gọi "C++". Bjarne Stroustrup mô tả ngôn ngữ C++ lần đầu tiên trong bài báo có nhan đề "Data Abstraction in C". Sau một vài hiệu chỉnh C++ được công bố rộng rãi trong quyển "The C++ Programming Language" của Bjarne Stroustrup xuất hiện đánh dấu sự hiện diện thực sự của C++, người lập tình chuyên nghiệp từ đây đã có một ngôn ngữ đủ mạnh cho các dữ án thực tiễn của mình.

Về thực chất C++ giống như C nhưng bổ sung thêm một số mở rộng quan trọng, đặc biệt là ý tưởng về đối tượng, lập trình định hướng đối tượng.Thật ra các ý tưởng về cấu trúc trong C++ đã xuất phát vào các năm 1970 từ Simula 70 và Algol 68. Các ngôn ngữ này đã đưa ra các khái niệm về lớp và đơn thể. Ada là một ngôn ngữ phát triển từ đó, nhưng C++ đã khẳng định vai trò thực sự của mình.

II. CÁC MỞ RỘNG CỦA C++

II.1. Các từ khóa mới của C++

Để bổ sung các tính năng mới vào C, một số từ khóa (keyword) mới đã được đưa vào C++ ngoài các từ khóa có trong C. Các chương trình bằng C nào sử dụng các tên trùng với các từ khóa cần phải thay đổi trước khi chương trình được dịch lại bằng C++. Các từ khóa mới này là :

asm	**catch**	**class**	**delete**	**friend**	**inline**
new	**operator**	**private**	**protected**	**public**	**template**
this	**throw**	**try**	**virtual**		

II.2. Cách ghi chú thích

C++ chấp nhận hai kiểu chú thích. Các lập trình viên bằng C đã quen với cách chú thích bằng /*…*/. Trình biên dịch sẽ bỏ qua mọi thứ nằm giữa /*…*/.

Ví dụ 2.1: Trong chương trình sau :

```
#include <iostream.h>
int main()
{
      int I;
      for(I = 0; I < 10 ; ++ I) // 0 - 9
            cout<<I<<"\n";    // In ra
      return 0;
}
```

Mọi thứ nằm giữa /*…*/ từ dòng 1 đến dòng 3 đều được chương trình bỏ qua. Chương trình này còn minh họa cách chú thích thứ hai. Đó là cách chú thích bắt đầu bằng // ở dòng 8 và dòng 9. Chúng ta chạy ví dụ 2.1, kết quả ở hình 2.1.

Hình 2.1: Kết quả của ví dụ 2.1

Nói chung, kiểu chú thích /*…*/ được dùng cho các khối chú thích lớn gồm nhiều dòng, còn kiểu // được dùng cho các chú thích một dòng.

II.3. Dòng nhập/xuất chuẩn

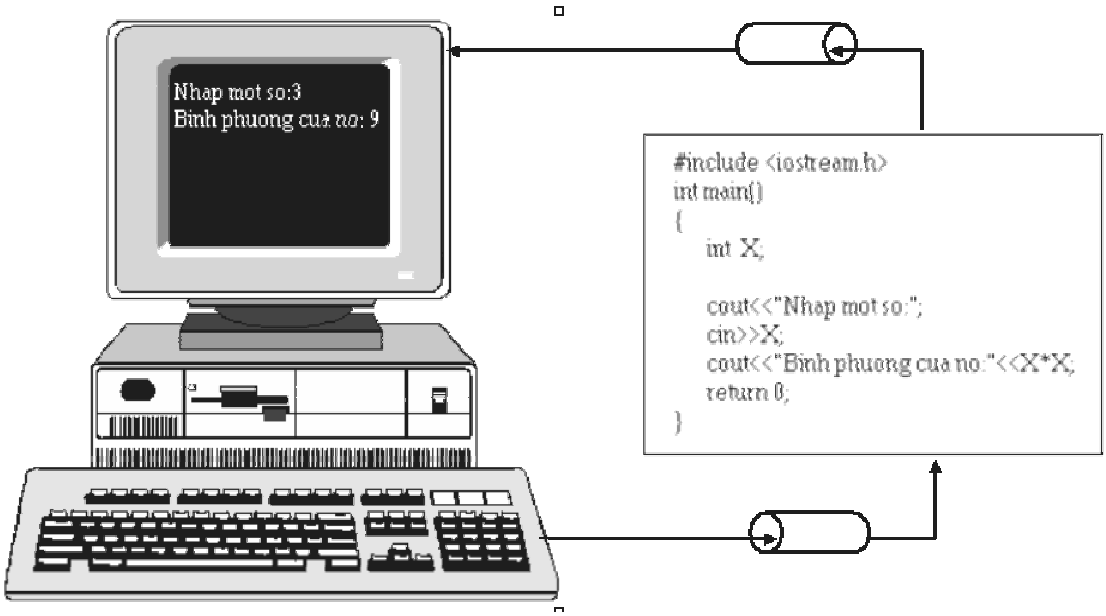
Trong chương trình C, chúng ta thường sử dụng các hàm nhập/xuất dữ liệu là printf() và scanf(). Trong C++ chúng ta có thể dùng dòng nhập/xuất chuẩn (standard input/output stream) để nhập/xuất dữ liệu thông qua hai biến đối tượng của dòng (stream object) là **cout** và **cin**.

Ví dụ 2.2: Chương trình nhập vào hai số. Tính tổng và hiệu của hai số vừa nhập.

```
//Chuong trinh 2.2
#include <iostream.h>
int main()
{
   int X, Y;
   cout<< "Nhap vao mot so X:";
   cin>>X;
   cout<< "Nhap vao mot so Y:";
   cin>>Y;
   cout<<"Tong cua chung:"<<X+Y<<"\n";
   cout<<"Hieu cua chung:"<<X-Y<<"\n";
   return 0;
}
```

Để thực hiện dòng xuất chúng ta sử dụng biến **cout** (console output) kết hợp với toán tử chèn (insertion operator) **<<** như ở các dòng 5, 7, 9 và 10. Còn dòng nhập chúng ta sử dụng biến **cin** (console input) kết hợp với toán tử trích (extraction operator) **>>** như ở các dòng 6 và 8. Khi sử dụng **cout** hay **cin**, chúng ta phải kéo file iostream.h như dòng 1. Chúng ta sẽ tìm hiểu kỹ về dòng nhập/xuất ở chương 8. Chúng ta chạy ví dụ 2.2 , kết quả ở hình 2.2.

Hình 2.2: Kết quả của ví dụ 2.2



Hình 2.3: Dòng nhập/xuất dữ liệu

II.4. Cách chuyển đổi kiểu dữ liệu

Hình thức chuyển đổi kiểu trong C tương đối tối nghĩa, vì vậy C++ trang bị thêm một cách chuyển đổi kiểu giống như một lệnh gọi hàm.

Ví dụ 2.3:

```
#include <iostream.h>
int main()
{
   int X = 200;
   long Y = (long) X; //Chuyen doi kieu theo cach cua C
   long Z = long(X);  //Chuyen doi kieu theo cach moi cua C++
   cout<< "X = "<<X<<"\n";
   cout<< "Y = "<<Y<<"\n";
   cout<< "Z = "<<Z<<"\n";

   return 0;
}
```

Chúng ta chạy ví dụ 2.3 , kết quả ở hình 2.4.

```
X = 200
Y = 200
Z = 200
```

Hình 2.4: Kết quả của ví dụ 2.3

II.5. Vị trí khai báo biến

Trong chương trình C đòi hỏi tất cả các khai báo bên trong một phạm vi cho trước phải được đặt ở ngay đầu của phạm vi đó. Điều này có nghĩa là tất cả các khai báo toàn cục phải đặt trước tất cả các hàm và các khai báo cục bộ phải được tiến hành trước tất cả các lệnh thực hiện. Ngược lại C++ cho phép chúng ta khai báo linh hoạt bất kỳ vị trí nào trong một phạm vi cho trước (không nhất thiết phải ngay đầu của phạm vi), chúng ta xen kẽ việc khai báo dữ liệu với các câu lệnh thực hiện.

Ví dụ 2.4: Chương trình mô phỏng một máy tính đơn giản

```
1: #include <iostream.h>
2: int main()
3: {
4:    int X;
```

```
5:    cout<< "Nhap vao so thu nhat:";
6:    cin>>X;
7:    int Y;
8:    cout<< "Nhap vao so thu hai:";
9:    cin>>Y;
10:    char Op;
11:    cout<<"Nhap vao toan tu (+-*/):";
12:    cin>>Op;
13:    switch(Op)
14:    {
15:       case '+':
16:          cout<<"Ket qua:"<<X+Y<<"\n";
17:          break;
18:       case '-':
19:          cout<<"Ket qua:"<<X-Y<<"\n";
20:          break;
21:       case '*':
22:          cout<<"Ket qua:"<<long(X)*Y<<"\n";
23:          break;
24:       case '/':
25:          if (Y)
26:             cout<<"Ket qua:"<<float(X)/Y<<"\n";
27:          else
28:             cout<<"Khong the chia duoc!" <<"\n"; 9; 9;
29:          break;
30:       default :
31:          cout<<"Khong hieu toan tu nay!"<<"\n";
32:    }
33:    return 0;
34: }
```

Trong chương trình chúng ta xen kẻ khai báo biến với lệnh thực hiện ở dòng 4 đến dòng 12. Chúng ta chạy ví dụ 2.4, kết quả ở hình 2.5.

```
Nhap vao so thu nhat:5
Nhap vao so thu hai:9
Nhap vao toan tu (+-*/):*
Ket qua:45
```

Hình 2.5: Kết quả của ví dụ 2.4

Khi khai báo một biến trong chương trình, biến đó sẽ có hiệu lực trong phạm vi của chương trình đó kể từ vị trí nó xuất hiện. Vì vậy chúng ta không thể sử dụng một biến được khai báo bên dưới nó.

II.6. Các biến const

Trong ANSI C, muốn định nghĩa một hằng có kiểu nhất định thì chúng ta dùng biến **const** (vì nếu dùng **#define** thì tạo ra các hằng không có chứa thông tin về kiểu). Trong C++, các biến **const** linh hoạt hơn một cách đáng kể:

C++ xem **const** cũng như **#define** nếu như chúng ta muốn dùng hằng có tên trong chương trình. Chính vì vậy chúng ta có thể dùng **const** để quy định kích thước của một mảng như đoạn mã sau:

```
const int ArraySize = 100;

int X[ArraySize];
```

Khi khai báo một biến **const** trong C++ thì chúng ta phải khởi tạo một giá trị ban đầu nhưng đối với ANSI C thì không nhất thiết phải làm như vậy (vì trình biên dịch ANSI C tự động gán trị zero cho biến **const** nếu chúng ta không khởi tạo giá trị ban đầu cho nó).

Phạm vi của các biến **const** giữa ANSI C và C++ khác nhau. Trong ANSI C, các biến **const** được khai báo ở bên ngoài mọi hàm thì chúng có phạm vi toàn cục, điều này nghĩa là chúng có thể nhìn thấy cả ở bên ngoài file mà chúng được định nghĩa, trừ khi chúng được khai báo là **static**. Nhưng trong C++, các biến **const** được hiểu mặc định là **static**.

II.7. Về struct, union và enum

Trong C++, các **struct** và **union** thực sự các các kiểu **class**. Tuy nhiên có sự thay đổi đối với C++. Đó là tên của **struct** và **union** được xem luôn là tên kiểu giống như khai báo bằng lệnh **typedef** vậy.

Trong C, chúng ta có thể có đoạn mã sau :	Trong C++, vấn đề trở nên đơn giản hơn:

struct Complex
{
float Real;
float Imaginary;
};
……..
struct Complex C;
	struct Complex
	{
	float Real;
	float Imaginary;
	};
	……..
	Complex C;

Quy định này cũng áp dụng cho cả **union** và **enum**. Tuy nhiên để tương thích với C, C++ vẫn chấp nhận cú pháp cũ.

Một kiểu **union** đặc biệt được thêm vào C++ gọi là union nặc danh (anonymous union). Nó chỉ khai báo một loạt các trường(field) dùng chung một vùng địa chỉ bộ nhớ. Một union nặc danh không có tên tag, các trường có thể được truy xuất trực tiếp bằng tên của chúng. Chẳng hạn như đoạn mã sau:

```
union
{
    int Num;
    float Value;
};
```

Cả hai Num và Value đều dùng chung một vị trí và không gian bộ nhớ. Tuy nhiên không giống như kiểu **union** có tên, các trường của union nặc danh thì được truy xuất trực tiếp, chẳng hạn như sau:

Num = 12;

Value = 30.56;

II.8. Toán tử định phạm vi

Toán tử định phạm vi (scope resolution operator) ký hiệu là **::**, nó được dùng truy xuất một phần tử bị che bởi phạm vi hiện thời.

Ví dụ 2.5 :

```
1: #include <iostream.h>
2: int X = 5;
3: int main()
4: {
5:     int X = 16;
6:     cout<< "Bien X ben trong = "<<X<<"\n";
7:     cout<< "Bien X ben ngoai = "<<::X<<"\n";
8:     return 0;
9: }
```

Chúng ta chạy ví dụ 2.5, kết quả ở hình 2.6

```
Bien X ben trong = 16
Bien X ben ngoai = 5
```

Hình 2.6: Kết quả của ví dụ 2.5

Toán tử định phạm vi còn được dùng trong các định nghĩa hàm của các phương thức trong các lớp, để khai báo lớp chủ của các phương thức đang được định nghĩa đó. Toán tử định phạm vi còn có thể được dùng để phân biệt các thành phần trùng tên của các lớp cơ sở khác nhau.

II.9. Toán tử new và delete

Trong các chương trình C, tất cả các cấp phát động bộ nhớ đều được xử lý thông qua các hàm thư viện như **malloc()**, **calloc()** và **free()**. C++ định nghĩa một phương thức mới để thực hiện việc cấp phát động bộ nhớ bằng cách dùng hai toán tử **new** và **delete**. Sử dụng hai toán tử này sẽ linh hoạt hơn rất nhiều so với các hàm thư viện của C.

Đoạn chương trình sau dùng để cấp phát vùng nhớ động theo lối cổ điển của C.

```
int *P;
P = malloc(sizeof(int));
if (P==NULL)
  printf("Khong con du bo nho de cap phat\n");
else
{
  *P = 290;
  printf("%d\n", *P);
  free(P);
}
```

Trong C++, chúng ta có thể viết lại đoạn chương trình trên như sau:

```
int *P;
P = new int;
if (P==NULL)
  cout<<"Khong con du bo nho de cap phat\n";
else
  {
  *P = 290;
  cout<<*P<<"\n";
  delete P;
  }
```

Chúng ta nhận thấy rằng, cách viết của C++ sáng sủa và dễ sử dụng hơn nhiều. Toán tử **new** thay thế cho hàm **malloc()** hay **calloc()** của C có cú pháp như sau :

new *type_name*

new (*type_name* **)**

new *type_name initializer*

new (*type_name* **)** *initializer*

Trong đó :

type_name: Mô tả kiểu dữ liệu được cấp phát. Nếu kiểu dữ liệu mô tả phức tạp, nó có thể được đặt bên trong các dấu ngoặc.

initializer: Giá trị khởi động của vùng nhớ được cấp phát.

Nếu toán tử **new** cấp phát không thành công thì nó sẽ trả về giá trị **NULL**.

Còn toán tử **delete** thay thế hàm **free()** của C, nó có cú pháp như sau :

delete *pointer*

delete [] *pointer*

Chúng ta có thể vừa cấp phát vừa khởi động như sau :

```
int *P;
P = new int(100);
if (P!=NULL)
{
    cout<<*P<<"\n";
    delete P;
}
else
    cout<<"Khong con du bo nho de cap phat\n";
```

Để cấp phát một mảng, chúng ta làm như sau :

```
int *P;
P = new int[10]; //Cấp phát mảng 10 số nguyên
if (P!=NULL)
{
for(int I = 0;I<10;++)
      P[I]= I;
for(I = 0;I<10;++)
      cout<<P[I]<<"\n";
delete []P;
}else cout<<"Khong con du bo nho de cap phat\n";
```

Chú ý: Đối với việc cấp phát mảng chúng ta không thể vừa cấp phát vừa khởi động giá trị cho chúng, chẳng hạn đoạn chương trình sau là sai :

```
int *P;
P = new (int[10])(3); //Sai !!!
```

Ví dụ 2.6: Chương trình tạo một mảng động, khởi động mảng này với các giá trị ngẫu nhiên và sắp xếp chúng.

```
1: #include <iostream.h>
2: #include <time.h>
3: #include <stdlib.h>
4: int main()
5: {
6:    int N;
7:    cout<<"Nhap vao so phan tu cua mang:";
8:    cin>>N;
9:    int *P=new int[N];
10:    if (P==NULL)
11:    {
12:       cout<<"Khong con bo nho de cap phat\n";
13:       return 1;
14:    }
15:    srand((unsigned)time(NULL));
16:    for(int I=0;I<N;++I)
17:      P[I]=rand()%100; //Tạo các số ngẫu nhiên từ 0 đến 99
18:    cout<<"Mang truoc khi sap xep\n";
19:    for(I=0;I<N;++I)
20:       cout<<P[I]<<" ";
21:    for(I=0;I<N-1;++I)
22:       for(int J=I+1;J<N;++J)
23:          if (P[I]>P[J])
24:          {
25:             int Temp=P[I];
26:             P[I]=P[J];
27:             P[J]=Temp;
28:          }
29:    cout<<"\nMang sau khi sap xep\n";
30:    for(I=0;I<N;++I)
31:       cout<<P[I]<<" ";
32:    delete []P;
33:    return 0;
34: }
```

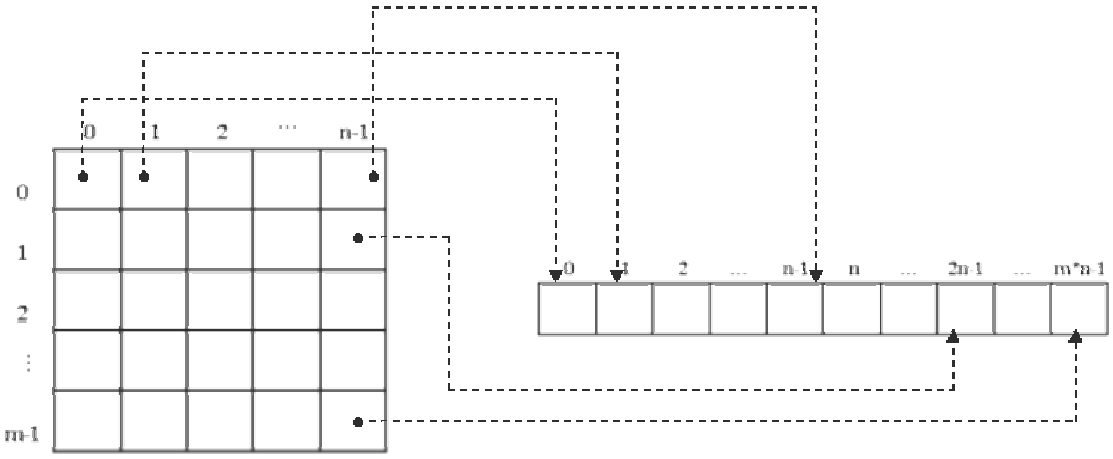
Chúng ta chạy ví dụ 2.6, kết quả ở hình 2.7

```
Nhap vao so phan tu cua mang: 10
Mang truoc khi sap xep
81 83 11 71 58 7 5 59 74 24
Mang sau khi sap xep
5 7 11 24 58 59 71 74 81 83
```

Hình 2.7: Kết quả của ví dụ 2.6

Ví dụ 2.7: Chương trình cộng hai ma trận trong đó mỗi ma trận được cấp phát động.

Chúng ta có thể xem mảng hai chiều như mảng một chiều như hình 2.8



Hình 2.8: Mảng hai chiều có thể xem như mảng một chiều.

Gọi X là mảng hai chiều có kích thước m dòng và n cột.

A là mảng một chiều tương ứng.

Nếu X[i][j] chính là A[k] thì k = i*n + j

Chúng ta có chương trình như sau :

```
1: #include <iostream.h>
2: #include <conio.h>
3: //prototype
4: void AddMatrix(int * A,int *B,int*C,int M,int N);
5: int AllocMatrix(int **A,int M,int N);
6: void FreeMatrix(int *A);
7: void InputMatrix(int *A,int M,int N,char Symbol);
8: void DisplayMatrix(int *A,int M,int N);
9:
10: int main()
11: {
12:     int M,N;
13:     int *A = NULL,*B = NULL,*C = NULL;
14:
15:     clrscr();
16:     cout<<"Nhap so dong cua ma tran:";
17:     cin>>M;
18:     cout<<"Nhap so cot cua ma tran:";
19:     cin>>N;
20:     //Cấp phát vùng nhớ cho ma trận A
21:     if (!AllocMatrix(&A,M,N))
22:     { //endl: Xuất ra kí tự xuống dòng ('\n')
23:         cout<<"Khong con du bo nho!"<<endl;
24:         return 1;
25:     }
```

```
26:     //Cấp phát vùng nhớ cho ma trận B
27:     if (!AllocMatrix(&B,M,N))
28:     {
29:         cout<<"Khong con du bo nho!"<<endl;
30:         FreeMatrix(A);//Giải phóng vùng nhớ A
31:         return 1;
32:     }
33:     //Cấp phát vùng nhớ cho ma trận C
34:     if (!AllocMatrix(&C,M,N))
35:     {
36:         cout<<"Khong con du bo nho!"<<endl;
37:         FreeMatrix(A);//Giải phóng vùng nhớ A
38:         FreeMatrix(B);//Giải phóng vùng nhớ B
39:         return 1;
40:     }
41:     cout<<"Nhap ma tran thu 1"<<endl;
42:     InputMatrix(A,M,N,'A');
43:     cout<<"Nhap ma tran thu 2"<<endl;
44:     InputMatrix(B,M,N,'B');
45:     clrscr();
46:     cout<<"Ma tran thu 1"<<endl;
47:     DisplayMatrix(A,M,N);
48:     cout<<"Ma tran thu 2"<<endl;
49:     DisplayMatrix(B,M,N);
50:     AddMatrix(A,B,C,M,N);
51:     cout<<"Tong hai ma tran"<<endl;
52:     DisplayMatrix(C,M,N);
53:     FreeMatrix(A);//Giải phóng vùng nhớ A
54:     FreeMatrix(B);//Giải phóng vùng nhớ B
55:     FreeMatrix(C);//Giải phóng vùng nhớ C
56:     return 0;
57: }
68: //Cộng hai ma trận
69: void AddMatrix(int *A,int *B,int*C,int M,int N)
70: {
71:     for(int I=0;I<M*N;++I)
72:     C[I] = A[I] + B[I];
73: }
74: //Cấp phát vùng nhớ cho ma trận
75: int AllocMatrix(int **A,int M,int N)
76: {
77:     *A = new int [M*N];
78:     if (*A == NULL)
79:         return 0;
80:     return 1;
81: }
82: //Giải phóng vùng nhớ
83: void FreeMatrix(int *A)
84: {
85:     if (A!=NULL)
86:         delete [] A;
87: }
88: //Nhập các giá trị của ma trận
89: void InputMatrix(int *A,int M,int N,char Symbol)
90: {
91:     for(int I=0;I<M;++I)
92:     for(int J=0;J<N;++J)
93     {
```

```
94:          cout<<Symbol<<"["<<I<<"]["<<J<<"]=";
95:          cin>>A[I*N+J];
96:      }
97: }
100: //Hiển thị ma trận
101: void DisplayMatrix(int *A,int M,int N)
102: {
103:    for(int I=0;I<M;++I)
104:    {
105:      for(int J=0;J<N;++J)
106:     {
107:      out.width(7);//canh le phai voi chieu dai 7 ky tu
108:      cout<<A[I*N+J];
109:     }
110:    cout<<endl;
111:    }
112: }
```

Chúng ta chạy ví dụ 2.7 , kết quả ở hình 2.9

```
Nhap so dong cua ma tran:2
Nhap so cot cua ma tran:3
Nhap ma tran thu 1
A[0][0]=1
A[0][1]=2
A[0][2]=3
A[1][0]=4
A[1][1]=5
A[1][2]=6
Nhap ma tran thu 2
B[0][0]=7
B[0][1]=8
B[0][2]=9
B[1][0]=10
B[1][1]=11
B[1][2]=12
```

```
Ma tran thu 1
   1   2   3
   4   5   6
Ma tran thu 2
   7   8   9
  10  11  12
Tong hai ma tran
   8  10  12
  14  16  18
```

Hình 2.9: Kết quả của ví dụ 2.7

Một cách khác để cấp phát mảng hai chiều *A* gồm *M* dòng và *N* cột như sau:

```
int ** A = new int *[M];
int * Tmp = new int[M*N];
for(int I=0;I<M;++I)
{
        A[I]=Tmp;
        Tmp+=N;
}
//Thao tác trên mảng hai chiều A
……..
delete [] *A;
delete [] A;
```

Toán tử **new** còn có một thuận lợi khác, đó là tất cả các lỗi cấp phát động đều có thể bắt được bằng một hàm xử lý lỗi do người dùng tự định nghĩa. C++ có định nghĩa một con trỏ (pointer) trỏ đến hàm đặc biệt.

Khi toán tử **new** được sử dụng để cấp phát động và một lỗi xảy ra do cấp phát, C++ tự gọi đến hàm được chỉ bởi con trỏ này. Định nghĩa của con trỏ này như sau:

typedef void (*pvf)();

pvf _new_handler(pvf p);

Điều này có nghĩa là con trỏ **_new_handler** là con trỏ trỏ đến hàm không có tham số và không trả về giá trị. Sau khi chúng ta định nghĩa hàm như vậy và gán địa chỉ của nó cho **_new_handler** chúng ta có thể bắt được tất cả các lỗi do cấp phát động.

Ví dụ 2.8:

```
1: #include <iostream.h>
2: #include <stdlib.h>
3: #include <new.h>
4:
5: void MyHandler();
6:
7: unsigned long I = 0; 9;
8: void main()
9: {
10:    int *A;
11:    _new_handler = MyHandler;
12:    for( ; ; ++I)
13:       A = new int;
14:
15: }
16:
17: void MyHandler()
18: {
19:    cout<<"Lan cap phat thu "<<I<<endl;
20:    cout<<"Khong con du bo nho!"<<endl;
21:    exit(1);
22: }
```

Sử dụng con trỏ **_new_handler** chúng ta phải include file new.h như ở dòng 3. Chúng ta chạy ví dụ 2.8, kết quả ở hình 2.10.

```
Lan cap phat thu 7704
Khong con du bo nho!
```

Hình 2.10: Kết quả của ví dụ 2.8

Thư viện cũng còn có một hàm được định nghĩa trong new.h là hàm có prototype sau :

void (* set_new_handler(void (* my_handler)()))();

Hàm set_new_handler() dùng để gán một hàm cho _new_handler.

Ví dụ 2.9:

```
1: #include <iostream.h>
2: #include <new.h>
3: #include <stdlib.h>
4:
5: void MyHandler();
6:
7: int main(void)
8: {
9:
10:    char *Ptr;
11:
12:    set_new_handler(MyHandler);
```

```
13:     Ptr = new char[64000u];
14:     set_new_handler(0); //Thiết lập lại giá trị mặc định
15:     return 0;
16: }
17:
18: void MyHandler()
19: {
20:     cout <<endl<<"Khong con du bo nho";
21:     exit(1);
22 }
```

Chúng ta chạy ví dụ 2.9, kết quả ở hình 2.11

Hình 2.11: Kết quả của ví dụ 2.9

II.10. Hàm inline

Một chương trình có cấu trúc tốt sử dụng các hàm để chia chương trình thành các đơn vị độc lập có logic riêng. Tuy nhiên, các hàm thường phải chứa một loạt các xử lý điểm vào (entry point): tham số phải được đẩy vào stack, một lệnh gọi phải được thực hiện và sau đó việc quay trở về cũng phải được thực hiện bằng cách giải phóng các tham số ra khỏi stack. Khi các xử lý điểm vào chậm chạp thường các lập trình viên C phải sử dụng cách chép lập lại các đoạn chương trình nếu muốn tăng hiệu quả.

Để tránh khỏi phải xử lý điểm vào, C++ trang bị thêm từ khóa **inline** để loại việc gọi hàm. Khi đó trình biên dịch sẽ không biên dịch hàm này như một đoạn chương trình riêng biệt mà nó sẽ được chèn thẳng vào các chỗ mà hàm này được gọi. Điều này làm giảm việc xử lý điểm vào mà vẫn cho phép một chương trình được tổ chức dưới dạng có cấu trúc. Cú pháp của hàm **inline** như sau :

inline data_type function_name **(** parameters **)**
{
……..
}

Trong đó: data_type: Kiểu trả về của hàm.

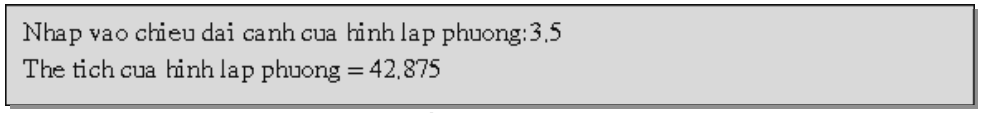
Function_name:Tên của hàm.

Parameters: Các tham số của hàm.

Ví dụ 2.10: Tính thể tích của hình lập phương

```
1: #include <iostream.h>
2: inline float Cube(float S)
3: {
4: return S*S*S;
5: }
6:
7: int main()
8: {
9:     cout<<"Nhap vao chieu dai canh cua hinh lap phuong:";
10:    float Side;
11:    cin>>Side;
12:    cout<<"The tich cua hinh lap phuong = "<<Cube(Side);
13:    return 0;
14: }
```

Chúng ta chạy ví dụ 2.10, kết quả ở hình 2.12



Hình 2.12: Kết quả của ví dụ 2.10

Chú ý:

- Sử dụng hàm **inline** sẽ làm cho chương trình lớn lên vì trình biên dịch chèn đoạn chương trình vào các chỗ mà hàm này được gọi. Do đó thường các hàm **inline** thường là các hàm nhỏ, ít phức tạp.

- Các hàm **inline** phải được định nghĩa trước khi sử dụng. Ở ví dụ 2.10 chúng ta sửa lại như sau thì chương trình sẽ bị báo lỗi:

```
#include <iostream.h>
float Cube(float S);
int main()
{
    cout<<"Nhap vao chieu dai canh cua hinh lap phuong:";
    float Side;
    cin>>Side;
    cout<<"The tich cua hinh lap phuong = "<<Cube(Side);
    return 0;
}
inline float Cube(float S)
{
   return S*S*S;
}
```

- Các hàm đệ quy không được là hàm inline.

II.11. Các giá trị tham số mặc định

Một trong các đặc tính nổi bật nhất của C++ là khả năng định nghĩa các giá trị tham số mặc định cho các hàm. Bình thường khi gọi một hàm, chúng ta cần gởi một giá trị cho mỗi tham số đã được định nghĩa trong hàm đó, chẳng hạn chúng ta có đoạn chương trình sau:

```
void MyDelay(long Loops); //prototype
……..
void MyDelay(long Loops)
{
for(int I = 0; I < Loops; ++I)
;
}
```

Mỗi khi hàm *MyDelay()* được gọi chúng ta phải gởi cho nó một giá trị cho tham số *Loops*. Tuy nhiên, trong nhiều trường hợp chúng ta có thể nhận thấy rằng chúng ta luôn luôn gọi hàm *MyDelay()* với cùng một giá trị *Loops* nào đó. Muốn vậy chúng ta sẽ dùng giá trị mặc định cho tham số *Loops*, giả sử chúng ta muốn giá trị mặc định cho tham số *Loops* là 1000. Khi đó đoạn mã trên được viết lại như sau :

```
void MyDelay(long Loops = 1000); //prototype
……..
void MyDelay(long Loops)
{
for(int I = 0; I < Loops; ++I)
;
}
```

Mỗi khi gọi hàm *MyDelay()* mà không gởi một tham số tương ứng thì trình biên dịch sẽ tự động gán cho tham số *Loops* giá trị 1000.

```
MyDelay(); // Loops có giá trị là 1000

MyDelay(5000); // Loops có giá trị là 5000
```

Giá trị mặc định cho tham số có thể là một hằng, một hàm, một biến hay một biểu thức.

Ví dụ 2.11: Tính thể tích của hình hộp

```
1: #include <iostream.h>
2: int BoxVolume(int Length = 1, int Width = 1, int Height = 1);
3:
4: int main()
5: {
6:   cout << "The tich hinh hop mac dinh: "
7:   << BoxVolume() << endl << endl
8:   << "The tich hinh hop voi chieu dai=10,do rong=1,chieu cao=1:"
9:   << BoxVolume(10) << endl << endl
10:  << "The tich hinh hop voi chieu dai=10,do rong=5,chieu cao=1:"
11:  << BoxVolume(10, 5) << endl << endl
12:  << "The tich hinh hop voi chieu dai=10,do rong=5,chieu cao=2:"
13:  << BoxVolume(10, 5, 2)<< endl;
14:    return 0;
15: }
16: //Tính thể tích của hình hộp
17: int BoxVolume(int Length, int Width, int Height)
18: {
19:    return Length * Width * Height;
20: }
```

Chúng ta chạy ví dụ 2.11, kết quả ở hình 2.13

```
The tich hinh hop mac dinh: 1

The tich hinh hop voi chieu dai=10,do rong=1,chieu cao=1: 10

The tich hinh hop voi chieu dai=10,do rong=5,chieu cao=1: 50

The tich hinh hop voi chieu dai=10,do rong=5,chieu cao=2: 100
```

Hình 2.13: Kết quả của ví dụ 2.11

⚠Chú ý:

- Các tham số có giá trị mặc định chỉ được cho trong prototype của hàm và không được lặp lại trong định nghĩa hàm (Vì trình biên dịch sẽ dùng các thông tin trong prototype chứ không phải trong định nghĩa hàm để tạo một lệnh gọi).

- Một hàm có thể có nhiều tham số có giá trị mặc định. Các tham số có giá trị mặc định cần phải được nhóm lại vào các tham số cuối cùng (hoặc duy nhất) của một hàm. Khi gọi hàm có nhiều tham số có giá trị mặc định, chúng ta chỉ có thể bỏ bớt các tham số theo thứ tự từ phải sang trái và phải bỏ liên tiếp nhau, chẳng hạn chúng ta có đoạn chương trình như sau:

int MyFunc(int a= 1, int b , int c = 3, int d = 4); //prototype sai!!!

int MyFunc(int a, int b = 2 , int c = 3, int d = 4); //prototype đúng

……..

MyFunc(); // Lỗi do tham số a không có giá trị mặc định

MyFunc(1);// OK, các tham số b, c và d lấy giá trị mặc định

MyFunc(5, 7); // OK, các tham số c và d lấy giá trị mặc định

MyFunc(5, 7, , 8); // Lỗi do các tham số bị bỏ phải liên tiếp nhau

II.12. Phép tham chiếu

Trong C, hàm nhận tham số là con trỏ đòi hỏi chúng ta phải thận trọng khi gọi hàm. Chúng ta cần viết hàm hoán đổi giá trị giữa hai số như sau:

```
void Swap(int *X, int *Y);
{
int Temp = *X;
*X = *Y;
*Y = *Temp;
}
```

Để hoán đổi giá trị hai biến *A* và *B* thì chúng ta gọi hàm như sau:

```
Swap(&A, &B);
```

Rõ ràng cách viết này không được thuận tiện lắm. Trong trường hợp này, C++ đưa ra một kiểu biến rất đặc biệt gọi là biến tham chiếu (reference variable). Một biến tham chiếu giống như là một bí danh của biến khác. Biến tham chiếu sẽ làm cho các hàm có thay đổi nội dung các tham số của nó được viết một cách thanh thoát hơn. Khi đó hàm *Swap()* được viết như sau:

```
void Swap(int &X, int &Y);
{
int Temp = X;
X = Y;
Y = Temp ;
}
```

Chúng ta gọi hàm như sau :

```
Swap(A, B);
```

Với cách gọi hàm này, C++ tự gởi địa chỉ của *A* và *B* làm tham số cho hàm *Swap()*. Cách dùng biến tham chiếu cho tham số của C++ tương tự như các tham số được khai báo là **Var** trong ngôn ngữ Pascal. Tham số này được gọi là tham số kiểu tham chiếu (reference parameter). Như vậy biến tham chiếu có cú pháp như sau :

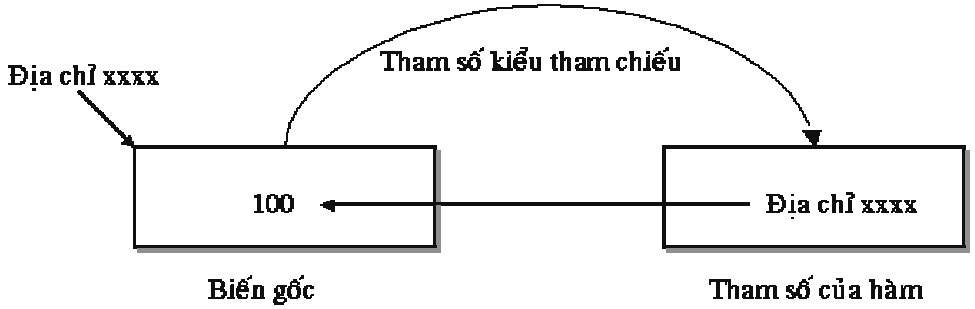
data_type & variable_name;

Trong đó:

data_type: Kiểu dữ liệu của biến.

variable_name: Tên của biến

Khi dùng biến tham chiếu cho tham số chỉ có địa chỉ của nó được gởi đi chứ không phải là toàn bộ cấu trúc hay đối tượng đó như hình 2.14, điều này rất hữu dụng khi chúng ta gởi cấu trúc và đối tượng lớn cho một hàm.



Hình 2.14: Một tham số kiểu tham chiếu nhận một tham chiếu
tới một biến được chuyển cho tham số của hàm.

Ví dụ 2.12: Chương trình hoán đổi giá trị của hai biến.

```
#include <iostream.h>
//prototype
void Swap(int &X,int &Y);
int main()
{
   int X = 10, Y = 5;
   cout<<"Truoc khi hoan doi: X = "<<X<<",Y = "<<Y<<endl;
   Swap(X,Y);
   cout<<"Sau khi hoan doi: X = "<<X<<",Y = "<<Y<<endl;
   return 0;
}
void Swap(int &X,int &Y)
{
    int Temp=X;
    X=Y;
    Y=Temp;
}
```

Chúng ta chạy ví dụ 2.12, kết quả ở hình 2.15

```
Truoc khi hoan doi: X = 10,Y = 5
Sau khi hoan doi: X = 5,Y = 10
```

Hình 2.15: Kết quả của ví dụ 2.12

Đôi khi chúng ta muốn gởi một tham số nào đó bằng biến tham chiếu cho hiệu quả, mặc dù chúng ta không muốn giá trị của nó bị thay đổi thì chúng ta dùng thêm từ khóa **const** như sau :

int MyFunc(const int & X);

Hàm *MyFunc()* sẽ chấp nhận một tham số *X* gởi bằng tham chiếu nhưng **const** xác định rằng *X* không thể bị thay đổi.

Biến tham chiếu có thể sử dụng như một bí danh của biến khác (bí danh đơn giản như một tên khác của biến gốc), chẳng hạn như đoạn mã sau :

int Count = 1;

int & Ref = Count; //Tạo biến Ref như là một bí danh của biến Count

++Ref; //Tăng biến Count lên 1 (sử dụng bí danh của biến Count)

Các biến tham chiếu phải được khởi động trong phần khai báo của chúng và chúng ta không thể gán lại một bí danh của biến khác cho chúng. Chẳng hạn đoạn mã sau là sai:

int X = 1;

int & Y; //Lỗi: Y phải được khởi động.

Khi một tham chiếu được khai báo như một bí danh của biến khác, mọi thao tác thực hiện trên bí danh chính là thực hiện trên biến gốc của nó. Chúng ta có thể lấy địa chỉ của biến tham chiếu và có thể so sánh các biến tham chiếu với nhau (phải tương thích về kiểu tham chiếu).

Ví dụ 2.13: Mọi thao tác trên trên bí danh chính là thao tác trên biến gốc của nó.

```
#include <iostream.h>
int main()
{
   int X = 3;
   int &Y = X; //Y la bí danh của X
   int Z = 100;
   cout<<"X="<<X<<endl<<"Y="<<Y<<endl;
   Y *= 3;
```

```
    cout<<"X="<<X<<endl<<"Y="<<Y<<endl;
    Y = Z;
    cout<<"X="<<X<<endl<<"Y="<<Y<<endl;
    return 0;
}
```

Chúng ta chạy ví dụ 2.13, kết quả ở hình 2.16

```
X=3
Y=3
X=9
Y=9
X=100
Y=100
```

Hình 2.16: Kết quả của ví dụ 2.13

Ví dụ 2.14: Lấy địa chỉ của biến tham chiếu

```
#include <iostream.h>
int main()
{
    int X = 3;
    int &Y = X; //Y la bí danh của X
    cout<<"Dia chi cua X = "<<&X<<endl;
    cout<<"Dia chi cua bi danh Y= "<<&Y<<endl;
    return 0;
}
```

Chúng ta chạy ví dụ 2.14, kết quả ở hình 2.17

```
Dia chi cua X=0xfff4
Dia chi cua bi danh Y=0xfff4
```

Hình 2.17: Kết quả của ví dụ 2.14

Chúng ta có thể tạo ra biến tham chiếu với việc khởi động là một hằng, chẳng hạn như đoạn mã sau :

```
int & Ref = 45;
```

Trong trường hợp này, trình biên dịch tạo ra một biến tạm thời chứa trị hằng và biến tham chiếu chính là bí danh của biến tạm thời này. Điều này gọi là tham chiếu độc lập (independent reference).

Các hàm có thể trả về một tham chiếu, nhưng điều này rất nguy hiểm. Khi hàm trả về một tham chiếu tới một biến cục bộ của hàm thì biến này phải được khai báo là **static**, ngược lại tham chiếu tới nó thì khi hàm kết thúc biến cục bộ này sẽ bị bỏ qua. Chẳng hạn như đoạn chương trình sau:

```
int & MyFunc()
{
static int X = 200; //Nếu không khai báo là static thì điều này rất nguy hiểm.
return X;
}
```

Khi một hàm trả về một tham chiếu, chúng ta có thể gọi hàm ở phía bên trái của một phép gán.

Ví dụ 2.15:

```
1: #include <iostream.h>
2:
3: int X = 4;
4: //prototype
```

```
5: int & MyFunc();
6:
7: int main()
8: {
9:     cout<<"X="<<X<<endl;
10:    cout<<"X="<<MyFunc()<<endl;
11:    MyFunc() = 20; //Nghĩa là X = 20
12:    cout<<"X="<<X<<endl;
13:    return 0;
14: }
15:
16: int & MyFunc()
17: {
18:    return X;
19: }
```

Chúng ta chạy ví dụ 2.15, kết quả ở hình 2.18

```
X=4
X=4
X=20
```

Hình 2.18: Kết quả của ví dụ 2.15

Chú ý:

- Mặc dù biến tham chiếu trông giống như là biến con trỏ nhưng chúng không thể là biến con trỏ do đó chúng không thể được dùng cấp phát động.
- Chúng ta không thể khai báo một biến tham chiếu chỉ đến biến tham chiếu hoặc biến con trỏ chỉ đến biến tham chiếu. Tuy nhiên chúng ta có thể khai báo một biến tham chiếu về biến con trỏ như đoạn mã sau:

```
int X;
int *P = &X;
int * & Ref = P;
```

II.13. Phép đa năng hóa (Overloading)

Với ngôn ngữ C++, chúng ta có thể đa năng hóa các hàm và các toán tử (operator). Đa năng hóa là phương pháp cung cấp nhiều hơn một định nghĩa cho tên hàm đã cho trong cùng một phạm vi. Trình biên dịch sẽ lựa chọn phiên bản thích hợp của hàm hay toán tử dựa trên các tham số mà nó được gọi.

II.13.1. Đa năng hóa các hàm (Functions overloading)

Trong ngôn ngữ C cũng như mọi ngôn ngữ máy tính khác, mỗi hàm đều phải có một tên phân biệt. Đôi khi đây là một điều phiền toái. Chẳng hạn như trong ngôn ngữ C, có rất nhiều hàm trả về trị tuyệt đối của một tham số là số, vì cần thiết phải có tên phân biệt nên C phải có hàm riêng cho mỗi kiểu dữ liệu số, do vậy chúng ta có tới ba hàm khác nhau để trả về trị tuyệt đối của một tham số:

```
int abs(int i);
long labs(long l);
double fabs(double d);
```

Tất cả các hàm này đều cùng thực hiện một chứa năng nên chúng ta thấy điều này nghịch lý khi phải có ba tên khác nhau. C++ giải quyết điều này bằng cách cho phép chúng ta tạo ra các hàm khác nhau có cùng một tên. Đây chính là đa năng hóa hàm. Do đó trong C++ chúng ta có thể định nghĩa lại các hàm trả về trị tuyệt đối để thay thế các hàm trên như sau :

```
int abs(int i);
long abs(long l);
double abs(double d);
```

Ví dụ 2.16:

```
1: #include <iostream.h>
2: #include <math.h>
3:
4: int MyAbs(int X);
5: long MyAbs(long X);
6: double MyAbs(double X);
7:
8: int main()
9: {
10:    int X = -7;
11:    long Y = 200000l;
12:    double Z = -35.678;
13:    cout<<"Tri tuyet doi cua so nguyen (int) "<<X<<" la "
14:        <<MyAbs(X)<<endl;
15:    cout<<"Tri tuyet doi cua so nguyen (long int) "<<Y<<" la "
16:        <<MyAbs(Y)<<endl;
17:    cout<<"Tri tuyet doi cua so thuc "<<Z<<" la "
18:        <<MyAbs(Z)<<endl;
19:    return 0;
20: }
21:
22: int MyAbs(int X)
23: {
24:    return abs(X);
25: }
26:
27: long MyAbs(long X)
28: {
29:    return labs(X);
30: }
31:
32: double MyAbs(double X)
33: {
34:    return fabs(X);
35: }
```

Chúng ta chạy ví dụ 2.16 , kết quả ở hình 2.19

```
Tri tuyet doi cua so nguyen (int) -7 la 7
Tri tuyet doi cua so nguyen (long int) 200000 la 200000
Tri tuyet doi cua so thuc -35.678 la 35.678
```

Hình 2.19: Kết quả của ví dụ 2.16

Trình biên dịch dựa vào sự khác nhau về số các tham số, kiểu của các tham số để có thể xác định chính xác phiên bản cài đặt nào của hàm *MyAbs()* thích hợp với một lệnh gọi hàm được cho, chẳng hạn như:

MyAbs(-7); //Gọi hàm int MyAbs(int)

MyAbs(-7*l*); //Gọi hàm long MyAbs(long)

MyAbs(-7.5); //Gọi hàm double MyAbs(double)

Quá trình tìm được hàm được đa năng hóa cũng là quá trình được dùng để giải quyết các trường hợp nhập nhằng của C++. Chẳng hạn như nếu tìm thấy một phiên bản định nghĩa nào đó của một hàm được đa năng hóa mà có kiểu dữ liệu các tham số của nó trùng với kiểu các tham số đã gởi tới trong lệnh gọi hàm thì phiên bản hàm đó sẽ được gọi. Nếu không trình biên dịch C++ sẽ gọi đến phiên bản nào cho phép chuyển kiểu dễ dàng nhất.

```
MyAbs('c'); //Gọi int MyAbs(int)
MyAbs(2.34f); //Gọi double MyAbs(double)
```

Các phép chuyển kiểu có sẵn sẽ được ưu tiên hơn các phép chuyển kiểu mà chúng ta tạo ra (chúng ta sẽ xem xét các phép chuyển kiểu tự tạo ở chương 3).

Chúng ta cũng có thể lấy địa chỉ của một hàm đã được đa năng hóa sao cho bằng một cách nào đó chúng ta có thể làm cho trình biên dịch C++ biết được chúng ta cần lấy địa chỉ của phiên bản hàm nào có trong định nghĩa. Chẳng hạn như:

```
int (*pf1)(int);
long (*pf2)(long);
int (*pf3)(double);
pf1 = MyAbs; //Trỏ đến hàm int MyAbs(int)
pf2 = MyAbs; //Trỏ đến hàm long MyAbs(long)
pf3 = MyAbs; //Lỗi!!! (không có phiên bản hàm nào để đối sánh)
```

⚠Các giới hạn của việc đa năng hóa các hàm:

- Bất kỳ hai hàm nào trong tập các hàm đã đa năng phải có các tham số khác nhau.
- Các hàm đa năng hóa với danh sách các tham số cùng kiểu chỉ dựa trên kiểu trả về của hàm thì trình biên dịch báo lỗi. Chẳng hạn như, các khai báo sau là không hợp lệ:

```
void Print(int X);
int Print(int X);
```

Không có cách nào để trình biên dịch nhận biết phiên bản nào được gọi nếu giá trị trả về bị bỏ qua. Như vậy các phiên bản trong việc đa năng hóa phải có sự khác nhau ít nhất về kiểu hoặc số tham số mà chúng nhận được.

- Các khai báo bằng lệnh **typedef** không định nghĩa kiểu mới. Chúng chỉ thay đổi tên gọi của kiểu đã có. Chúng không ảnh hưởng tới cơ chế đa năng hóa hàm. Chúng ta hãy xem xét đoạn mã sau:

```
typedef char * PSTR;
void Print(char * Mess);
void Print(PSTR Mess);
```

Hai hàm này có cùng danh sách các tham số, do đó đoạn mã trên sẽ phát sinh lỗi.

- Đối với kiểu mảng và con trỏ được xem như đồng nhất đối với sự phân biệt khác nhau giữa các phiên bản hàm trong việc đa năng hóa hàm. Chẳng hạn như đoạn mã sau se phát sinh lỗi:

```
void Print(char * Mess);
void Print(char Mess[]);
```

Tuy nhiên, đối với mảng nhiều chiều thì có sự phân biệt giữa các phiên bản hàm trong việc đa năng hóa hàm, chẳng hạn như đoạn mã sau hợp lệ:

```
void Print(char Mess[]);
void Print(char Mess[][7]);
void Print(char Mess[][9][42]);
```

- **const** và các con trỏ (hay các tham chiếu) có thể dùng để phân biệt, chẳng hạn như đoạn mã sau hợp lệ:

```
void Print(char *Mess);
void Print(const char *Mess);
```

II.13.2. Đa năng hóa các toán tử (Operators overloading) :

Trong ngôn ngữ C, khi chúng ta tự tạo ra một kiểu dữ liệu mới, chúng ta thực hiện các thao tác liên quan đến kiểu dữ liệu đó thường thông qua các hàm, điều này trở nên không thoải mái.

Ví dụ 2.17: Chương trình cài đặt các phép toán cộng và trừ số phức

```
1: #include <stdio.h>
2: /* Định nghĩa số phức */
3: typedef struct
4: {
5:     double Real;
6:     double Imaginary;
7: }Complex;
8:
9:  Complex SetComplex(double R,double I);
10: Complex AddComplex(Complex C1,Complex C2);
11: Complex SubComplex(Complex C1,Complex C2);
12: void DisplayComplex(Complex C);
13:
14: int main(void)
15: {
16:    Complex C1,C2,C3,C4;
17:
18:    C1 = SetComplex(1.0,2.0);
19:    C2 = SetComplex(-3.0,4.0);
20:    printf("\nSo phuc thu nhat:");
21:    DisplayComplex(C1);
22:    printf("\nSo phuc thu hai:");
23:    DisplayComplex(C2);
24:    C3 = AddComplex(C1,C2); //Hơi bất tiện !!!
25:    C4 = SubComplex(C1,C2);
26:    printf("\nTong hai so phuc nay:");
27:    DisplayComplex(C3);
28:    printf("\nHieu hai so phuc nay:");
29:    DisplayComplex(C4);
30:    return 0;
31: }
32:
33: /* Đặt giá trị cho một số phức */
34: Complex SetComplex(double R,double I)
35: {
36:    Complex Tmp;
37:
38:    Tmp.Real = R;
39:    Tmp.Imaginary = I;
40:    return Tmp;
41: }
42: /* Cộng hai số phức */
43: Complex AddComplex(Complex C1,Complex C2)
44: {
45:    Complex Tmp;
46:
47:    Tmp.Real = C1.Real+C2.Real;
48:    Tmp.Imaginary = C1.Imaginary+C2.Imaginary;
49:    return Tmp;
```

```
50: }
51:
52: /* Trừ hai số phức */
53: Complex SubComplex(Complex C1,Complex C2)
54: {
55:    Complex Tmp;
56:
57:    Tmp.Real = C1.Real-C2.Real;
58:    Tmp.Imaginary = C1.Imaginary-C2.Imaginary;
59:    return Tmp;
60: }
61:
62: /* Hiển thị số phức */
63: void DisplayComplex(Complex C)
64: {
65:    printf("(%.1lf,%.1lf)",C.Real,C.Imaginary);
66: }
```

Chúng ta chạy ví dụ 2.17, kết quả ở hình 2.20

```
So phuc thu nhat:(1.0,2.0)
So phuc thu hai:(-3.0,4.0)
Tong hai so phuc nay:(-2.0,6.0)
Hieu hai so phuc nay:(4.0,-2.0)
```

Hình 2.20: Kết quả của ví dụ 2.17

Trong chương trình ở ví dụ 2.17, chúng ta nhận thấy với các hàm vừa cài đặt dùng để cộng và trừ hai số phức 1+2i và –3+4i; người lập trình hoàn toàn không thoải mái khi sử dụng bởi vì thực chất thao tác cộng và trừ là các toán tử chứ không phải là hàm. Để khắc phục yếu điểm này, trong C++ cho phép chúng ta có thể định nghĩa lại chức năng của các toán tử đã có sẵn một cách tiện lợi và tự nhiên hơn rất nhiều. Điều này gọi là đa năng hóa toán tử. Khi đó chương trình ở ví dụ 2.17 được viết như sau:

Ví dụ 2.18:

```
1: #include <iostream.h>
2: // Định nghĩa số phức
3: typedef struct
4: {
5:     double Real;
6:     double Imaginary;
7: }Complex;
8:
9:  Complex SetComplex(double R,double I);
10: void DisplayComplex(Complex C);
11: Complex operator + (Complex C1,Complex C2);
12: Complex operator - (Complex C1,Complex C2);
13:
14: int main(void)
15: {
16:    Complex C1,C2,C3,C4;
17:
18:    C1 = SetComplex(1.0,2.0);
19:    C2 = SetComplex(-3.0,4.0);
20:    cout<<"\nSo phuc thu nhat:";
21:    DisplayComplex(C1);
22:    cout<<"\nSo phuc thu hai:";
23:    DisplayComplex(C2);
24:    C3 = C1 + C2;
```

```
25:    C4 = C1 - C2;
26:    cout<<"\nTong hai so phuc nay:";
27:    DisplayComplex(C3);
28:    cout<<"\nHieu hai so phuc nay:";
29:    DisplayComplex(C4);
30:    return 0;
31: }
32:
33: //Đặt giá trị cho một số phức
34: Complex SetComplex(double R,double I)
35: {
36:    Complex Tmp;
37:
38:    Tmp.Real = R;
39:    Tmp.Imaginary = I;
40:    return Tmp;
41: }
42:
43: //Cộng hai số phức
44: Complex operator + (Complex C1,Complex C2)
45: {
46:    Complex Tmp;
47:
48:    Tmp.Real = C1.Real+C2.Real;
49:    Tmp.Imaginary = C1.Imaginary+C2.Imaginary;
50:    return Tmp;
51: }
52:
53: //Trừ hai số phức
54: Complex operator - (Complex C1,Complex C2)
55: {
56:    Complex Tmp;
57:
58:    Tmp.Real = C1.Real-C2.Real;
59:    Tmp.Imaginary = C1.Imaginary-C2.Imaginary;
60:    return Tmp;
61: }
62:
63: //Hiển thị số phức
64: void DisplayComplex(Complex C)
65: {
66:    cout<<"("<<C.Real<<","<<C.Imaginary<<")";
67: }
```

Chúng ta chạy ví dụ 2.18, kết quả ở hình 2.21

```
So phuc thu nhat:(1,2)
So phuc thu hai:(-3,4)
Tong hai so phuc nay:(-2,6)
Hieu hai so phuc nay:(4,-2)
```

Hình 2.21: Kết quả của ví dụ 2.18

Như vậy trong C++, các phép toán trên các giá trị kiểu số phức được thực hiện bằng các toán tử toán học chuẩn chứ không phải bằng các tên hàm như trong C. Chẳng hạn chúng ta có lệnh sau:

C4 = AddComplex(C3, SubComplex(C1,C2));

thì ở trong C++, chúng ta có lệnh tương ứng như sau:

C4 = C3 + C1 - C2;

Chúng ta nhận thấy rằng cả hai lệnh đều cho cùng kết quả nhưng lệnh của C++ thì dễ hiểu hơn. C++ làm được điều này bằng cách tạo ra các hàm định nghĩa cách thực hiện của một toán tử cho các kiểu dữ liệu tự định nghĩa. Một hàm định nghĩa một toán tử có cú pháp sau:

data_type **operator** *operator_symbol* (*parameters*)
{
……
}

Trong đó: *data_type*: Kiểu trả về.

operator_symbol: Ký hiệu của toán tử.

parameters: Các tham số (nếu có).

Trong chương trình ví dụ 2.18, toán tử + là toán tử gồm hai toán hạng (gọi là toán tử hai ngôi; toán tử một ngôi là toán tử chỉ có một toán hạng) và trình biên dịch biết tham số đầu tiên là ở bên trái toán tử, còn tham số thứ hai thì ở bên phải của toán tử. Trong trường hợp lập trình viên quen thuộc với cách gọi hàm, C++ vẫn cho phép bằng cách viết như sau:

C3 = operator + (C1,C2);

C4 = operator - (C1,C2);

Các toán tử được đa năng hóa sẽ được lựa chọn bởi trình biên dịch cũng theo cách thức tương tự như việc chọn lựa giữa các hàm được đa năng hóa là khi gặp một toán tử làm việc trên các kiểu không phải là kiểu có sẵn, trình biên dịch sẽ tìm một hàm định nghĩa của toán tử nào đó có các tham số đối sánh với các toán hạng để dùng. Chúng ta sẽ tìm hiểu kỹ về việc đa năng hóa các toán tử trong chương 4.

Các giới hạn của đa năng hóa toán tử:

- Chúng ta không thể định nghĩa các toán tử mới.
- Hầu hết các toán tử của C++ đều có thể được đa năng hóa. Các toán tử sau không được đa năng hóa là :

Toán tử	Ý nghĩa
::	Toán tử định phạm vi.
.*	Truy cập đến con trỏ là trường của **struct** hay thành viên của **class**.
.	Truy cập đến trường của **struct** hay thành viên của **class**.
?:	Toán tử điều kiện
sizeof	

và chúng ta cũng không thể đa năng hóa bất kỳ ký hiệu tiền xử lý nào.

- Chúng ta không thể thay đổi thứ tự ưu tiên của một toán tử hay không thể thay đổi số các toán hạng của nó.
- Chúng ta không thể thay đổi ý nghĩa của các toán tử khi áp dụng cho các kiểu có sẵn.
- Đa năng hóa các toán tử không thể có các tham số có giá trị mặc định.

Các toán tử có thể đa năng hoá:

+	-	*	/	%	^
!	=	<	>	+=	-=
^=	&=	|=	<<	>>	<<=
<=	>=	&&	||	++	--
()	[]	new	delete	&	|
~	*=	/=	%=	>>=	==
!=	,	->	->*		

Các toán tử được phân loại như sau :

- Các toán tử một ngôi : *** & ~ ! ++ -- sizeof (**data_type**)**

Các toán tử này được định nghĩa chỉ có một tham số và phải trả về một giá trị cùng kiểu với tham số của chúng. Đối với toán tử **sizeof** phải trả về một giá trị kiểu **size_t** (định nghĩa trong stddef.h)

Toán tử **(**data_type**)** được dùng để chuyển đổi kiểu, nó phải trả về một giá trị có kiểu là data_type.

- Các toán tử hai ngôi: *** / % + - >> << > <**

>= <= == != & | ^ && ||

Các toán tử này được định nghĩa có hai tham số.

- Các phép gán: **= += -= *= /= %= >>= <<= ^= |=**

Các toán tử gán được định nghĩa chỉ có một tham số. Không có giới hạn về kiểu của tham số và kiểu trả về của phép gán.

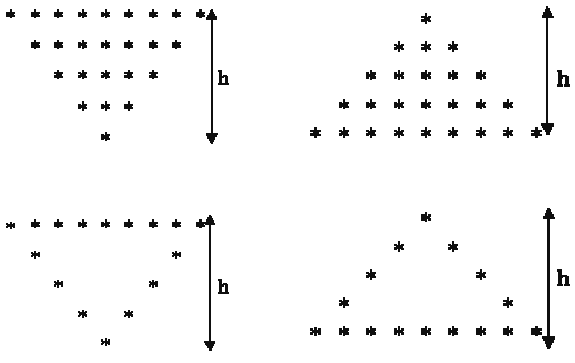
- Toán tử lấy thành viên : **->**
- Toán tử lấy phần tử theo chỉ số: **[]**
- Toán tử gọi hàm: **()**

BÀI TẬP

Bài 1: Hãy viết lại chương trình sau bằng cách sử dụng lại các dòng nhập/xuất trong C++.

```
/* Chương trình tìm mẫu chung nhỏ nhất */
#include <stdio.h>
int main()
{
      int a,b,i,min;
      printf("Nhap vao hai so:");
      scanf("%d%d",&a,&b);
      min=a>b?b:a;
      for(i = 2;i<min;++i)
        if (((a%i)==0)&&((b%i)==0)) break;
        if(i==min) {
          printf("Khong co mau chung nho nhat");
          return 0;
          }
      printf("Mau chung nho nhat la %d\n",i);
      return 0;
}
```

Bài 2: Viết chương trình nhập vào số nguyên dương h (2<h<23), sau đó in ra các tam giác có chiều cao là h như các hình sau:



Bài 3: Một tam giác vuông có thể có tất cả các cạnh là các số nguyên. Tập của ba số nguyên của các cạnh của một tam giác vuông được gọi là bộ ba Pitago. Đó là tổng bình phương của hai cạnh bằng bình phương của cạnh huyền, chẳng hạn bộ ba Pitago (3, 4, 5). Viết chương trình tìm tất cả các bộ ba Pitago như thế sao cho tất cả các cạnh không quá 500.

Bài 4: Viết chương trình in bảng của các số từ 1 đến 256 dưới dạng nhị phân, bát phân và thập lục phân tương ứng.

Bài 5: Viết chương trình nhập vào một số nguyên dương n. Kiểm tra xem số nguyên n có thuộc dãy Fibonacci không?

Bài 6: Viết chương trình nhân hai ma trân Amxn và Bnxp. Mỗi ma trận được cấp phát động và các giá trị của chúng phát sinh ngẫu nhiên (Với m, n và p nhập từ bàn phím).

Bài 7: Viết chương trình tạo một mảng một chiều động có kích thước là n (n nhập từ bàn phím). Các giá trị của mảng này được phát sinh ngẫu nhiên trên đoạn [a, b] với a và b đều nhập từ bàn phím. Hãy tìm số dương nhỏ nhất và số âm lớn nhất trong mảng; nếu không có số dương nhỏ nhất hoặc số âm lớn nhất thì xuất thông báo "không có số dương nhỏ nhất" hoặc "không có số âm lớn nhất".

Bài 8: Anh (chị) hãy viết một hàm tính bình phương của một số. Hàm sẽ trả về giá trị bình phương của tham số và có kiểu cùng kiểu với tham số.

Bài 9: Trong ngôn ngữ C, chúng ta có hàm chuyển đổi một chuỗi sang số, tùy thuộc vào dạng của chuỗi chúng ta có các hàm chuyển đổi sau :

- int atoi(const char *s);

  Chuyển đổi một chuỗi s thành số nguyên kiểu int.

- long atol(const char *s);

  Chuyển đổi một chuỗi s thành số nguyên kiểu long.

- double atof(const char *s);

  Chuyển đổi một chuỗi s thành số thực kiểu double.

Anh (chị) hãy viết một hàm có tên là aton (ascii to number) để chuyển đổi chuỗi sang các dạng số tương ứng.

Bài 10: Anh chị hãy viết các hàm sau:

- Hàm ComputeCircle() để tính diện tích s và chu vi c của một đường tròn bán kính r. Hàm này có prototype như sau:

  void ComputeCircle(float & s, float &c, float r = 1.0);

- Hàm ComputeRectangle() để tính diện tích s và chu vi p của một hình chữ nhật có chiều cao h và chiều rộng w. Hàm này có prototype như sau:

  void ComputeRectangle(float & s, float &p, float h = 1.0, float w = 1.0);

- Hàm ComputeTriangle() để tính diện tích s và chu vi p của một tam giác có ba cạnh a,b và c. Hàm này có prototype như sau:

  void ComputeTriangle(float & s, float &p, float a = 1.0, float b = 1.0, float c = 1.0);

- Hàm ComputeSphere() để tính thể tích v và diện tích bề mặt s của một hình cầu có bán kính r. Hàm này có prototype như sau:

  void ComputeSphere(float & v, float &s, float r = 1.0);

- Hàm ComputeCylinder() để tính thể tích v và diện tích bề mặt s của một hình trụ có bán kính r và chiều cao h. Hàm này có prototype như sau:

```
void ComputeCylinder(float & v, float &s, float r = 1.0 , float h = 1.0);
```

Bài 11: Anh (chị) hãy viết thêm hai toán tử nhân và chia hai số phức ở ví dụ 2.18 của chương 2.

Bài 12: Một cấu trúc Date chứa ngày, tháng và năm như sau:

```
struct Date
{
    int Day; //Có giá trị từ 1 → 31
    int Month; //Có giá trị từ 1 → 12
    int Year; //Biểu diễn bằng 4 chữ số.
};
```

Anh (chị) hãy viết các hàm định nghĩa các toán tử : **+ - > >= < <= == !=** trên cấu trúc Date này.

Bài 13: Một cấu trúc Point3D biểu diễn tọa độ của một điểm trong không gian ba chiều như sau:

```
struct Point3D
{
    float X;
    float Y;
    float Z;
};
```

Anh (chị) hãy viết các hàm định nghĩa các toán tử : **+ - == !=** trên cấu trúc Point3D này.

Bài 14: Một cấu trúc Fraction dùng để chứa một phân số như sau:

```
struct Fraction
{
    int Numerator; //Tử số
    int Denominator; //Mẫu số
};
```

Anh (chị) hãy viết các hàm định nghĩa các toán tử :

+ - * / > >= < <= == !=

trên cấu trúc Fraction này.

CHƯƠNG 3

LỚP VÀ ĐỐI TƯỢNG

I. DẪN NHẬP

Bây giờ chúng ta bắt đầu tìm hiểu về lập trình hướng đối tượng trong C++. Trong các phần sau, chúng ta cũng tìm hiểu về các kỹ thuật của thiết kế hướng đối tượng (Object-Oriented Design OOD): Chúng ta phân tích một vấn đề cụ thể, xác định các đối tượng nào cần để cài đặt hệ thống, xác định các thuộc tính nào mà đối tượng phải có, xác định hành vi nào mà đối tượng cần đưa ra, và chỉ rõ làm thế nào các đối tượng cần tương tác với đối tượng khác để thực hiện các mục tiêu tổng thể của hệ thống.

Chúng ta nhắc lại các khái niệm và thuật ngữ chính của định hướng đối tượng. OOP *đóng gói* dữ liệu (các thuộc tính) và các hàm (hành vi) thành gói gọi là *các đối tượng*. Dữ liệu và các hàm của đối tượng có sự liên hệ mật thiết với nhau. Các đối tượng có các đặc tính của việc che dấu thông tin. Điều này nghĩa là mặc dù các đối tượng có thể biết làm thế nào liên lạc với đối tượng khác thông qua các giao diện hoàn toàn xác định, bình thường các đối tượng không được phép biết làm thế nào các đối tượng khác được thực thi, các chi tiết của sự thi hành được dấu bên trong các đối tượng.

Trong C và các ngôn ngữ lập trình thủ tục, lập trình có khuynh hướng định hướng hành động, trong khi ý tưởng trong lập trình C++ là định hướng đối tượng. Trong C, đơn vị của lập trình là hàm; trong C++, đơn vị của lập trình là *lớp* (class) .

Các lập trình viên C tập trung vào viết các hàm. Các nhóm của các hành động mà thực hiện vài công việc được tạo thành các hàm, và các hàm được nhóm thành các chương trình. Dữ liệu thì rất quan trọng trong C, nhưng quan điểm là dữ liệu tồn tại chính trong việc hỗ trợ các hành động mà hàm thực hiện. Các động từ trong một hệ thống giúp cho lập trình viên C xác định tập các hàm mà sẽ hoạt động cùng với việc thực thi hệ thống.

Các lập trình viên C++ tập trung vào việc tạo ra "các kiểu do người dùng định nghĩa" (user-defined types) gọi là các lớp. Các lớp cũng được tham chiếu như "các kiểu do lập trình viên định nghĩa" (programmer-defined types). Mỗi lớp chứa dữ liệu cũng như tập các hàm mà xử lý dữ liệu. Các thành phần dữ liệu của một lớp được gọi là "các thành viên dữ liệu" (data members). Các thành phần hàm của một lớp được gọi là "các hàm thành viên" (member functions). Giống như thực thể của kiểu có sẵn như **int** được gọi là một biến, một thực thể của kiểu do người dùng định nghĩa (nghĩa là một lớp) được gọi là một đối tượng. Các danh từ trong một hệ thống giúp cho lập trình viên C++ xác định tập các lớp. Các lớp này được sử dụng để tạo các đối tượng mà sẽ sẽ hoạt động cùng với việc thực thi hệ thống.

Các lớp trong C++ được tiến hóa tự nhiên của khái niệm **struct** trong C. Trước khi tiến hành việc trình bày các lớp trong C++, chúng ta tìm hiểu về cấu trúc, và chúng ta xây dựng một kiểu do người dùng định nghĩa dựa trên một cấu trúc.

II. CÀI ĐẶT MỘT KIỂU DO NGƯỜI DÙNG ĐỊNH NGHĨA VỚI MỘT struct

Ví dụ 3.1: Chúng ta xây dựng kiểu cấu trúc *Time* với ba thành viên số nguyên: *Hour*, *Minute* và *second*. Chương trình định nghĩa một cấu trúc *Time* gọi là *DinnerTime*. Chương trình in thời gian dưới dạng giờ quân đội và dạng chuẩn.

```
#include <iostream.h>
struct Time
{
  int Hour;      // 0-23
  int Minute;    // 0-59
  int Second;    // 0-59
};
void PrintMilitary(const Time &);  //prototype
void PrintStandard(const Time &);  //prototype
int main()
{
  Time DinnerTime;
```

```
  //Thiet lap cac thanh vien voi gia tri hop le
  DinnerTime.Hour = 18;
  DinnerTime.Minute = 30;
  DinnerTime.Second = 0;
  cout << "Dinner will be held at ";
  PrintMilitary(DinnerTime);
  cout << " military time," << endl << "which is ";
  PrintStandard(DinnerTime);
  cout << " standard time." << endl;
  //Thiet lap cac thanh vien voi gia tri khong hop le
  DinnerTime.Hour = 29;
  DinnerTime.Minute = 73;
  DinnerTime.Second = 103;
  cout << endl << "Time with invalid values: ";
  PrintMilitary(DinnerTime);
  cout << endl;
  return 0;
}
//In thoi gian duoi dang gio quan doi
void PrintMilitary(const Time &T)
{
  cout << (T.Hour < 10 ? "0" : "") << T.Hour << ":"
       << (T.Minute < 10 ? "0" : "") << T.Minute << ":"
       << (T.Second < 10 ? "0" : "") << T.Second;
}
//In thoi gian duoi dang chuan
void PrintStandard(const Time &T)
{
  cout << ((T.Hour == 12) ? 12 : T.Hour % 12)
       << ":" << (T.Minute < 10 ? "0" : "") << T.Minute
       << ":" << (T.Second < 10 ? "0" : "") << T.Second
       << (T.Hour < 12 ? " AM" : " PM");
}
```

Chúng ta chạy ví dụ 3.1, kết quả ở hình 3.1

```
Dinner will be held at 18:30:00 military time,
which is 6:30:00 PM standard time.

Time with invalid values: 29:73:103
```

Hình 3.1: Kết quả của ví dụ 3.1

Có một vài hạn chế khi tạo các kiểu dữ liệu mới với các cấu trúc ở phần trên. Khi việc khởi tạo không được yêu cầu, có thể có dữ liệu chưa khởi tạo và các vấn đề nảy sinh. Ngay cả nếu dữ liệu được khởi tạo, nó có thể khởi tạo không chính xác. Các giá trị không hợp lệ có thể được gán cho các thành viên của một cấu trúc bởi vì chương trình trực tiếp truy cập dữ liệu. Chẳng hạn ở ví dụ 3.1 ở dòng 23 đến dòng 25, chương trình gán các giá trị không hợp lệ cho đối tượng *DinnerTime*. Nếu việc cài đặt của **struct** thay đổi, tất cả các chương trình sử dụng **struct** phải thay đổi. Điều này do lập trình viên trực tiếp thao tác kiểu dữ liệu. Không có "giao diện" để bảo đảm lập trình viên sử dụng dữ liệu chính xác và bảo đảm dữ liệu còn lại ở trạng thái thích hợp. Mặt khác, cấu trúc trong C không thể được in như một đơn vị, chúng được in khi các thành viên được in. Các cấu trúc trong C không thể so sánh với nhau, chúng phải được so sánh thành viên với thành viên.

Phần sau cài đặt lại cấu trúc *Time* ở ví dụ 3.1 như một lớp và chứng minh một số thuận lợi để việc tạo ra cái gọi là các kiểu dữ liệu trừu tượng (Abstract Data Types – ADT) như các lớp. Chúng ta sẽ thấy rằng các lớp và các cấu trúc có thể sử dụng gần như giống nhau trong C++. Sự khác nhau giữa chúng là thuộc tính truy cập các thành viên.

III. CÀI ĐẶT MỘT KIỂU DỮ LIỆU TRỪU TƯỢNG VỚI MỘT LỚP

Các lớp cho phép lập trình viên mô hình các đối tượng mà có các thuộc tính (biểu diễn như các thành viên dữ liệu – Data members) và các hành vi hoặc các thao tác (biểu diễn như các hàm thành viên – Member functions). Các kiểu chứa các thành viên dữ liệu và các hàm thành viên được định nghĩa thông thường trong C++ sử dụng từ khóa **class**, có cú pháp như sau:

class <class-name>

{

<member-list> //Thân của lớp

};

Trong đó:

class-name: tên lớp.

member-list: đặc tả các thành viên dữ liệu và các hàm thành viên.

Các hàm thành viên đôi khi được gọi là các phương thức (methods) trong các ngôn ngữ lập trình hướng đối tượng khác, và được đưa ra trong việc đáp ứng các message gởi tới một đối tượng. Một message tương ứng với việc gọi hàm thành viên.

Khi một lớp được định nghĩa, tên lớp có thể được sử dụng để khai báo đối tượng của lớp theo cú pháp sau:

<class-name> <object-name>;

Chẳng hạn, cấu trúc Time sẽ được định nghĩa dưới dạng lớp như sau:

```
class Time
{
    public:
    Time();
    void SetTime(int, int, int)
    void PrintMilitary();
    void PrintStandard()
    private:
    int Hour; // 0 - 23
    int Minute; // 0 - 59
    int Second; // 0 - 59
};
```

Trong định nghĩa lớp Time chứa ba thành viên dữ liệu là Hour, Minute và Second, và cũng trong lớp này, chúng ta thấy các nhãn **public** và **private** được gọi là các thuộc tính xác định truy cập thành viên (member access specifiers) gọi tắt là thuộc tính truy cập.

Bất kỳ thành viên dữ liệu hay hàm thành viên khai báo sau **public** có thể được truy cập bất kỳ nơi nào mà chương trình truy cập đến một đối tượng của lớp. Bất kỳ thành viên dữ liệu hay hàm thành viên khai báo sau **private** chỉ có thể được truy cập bởi các hàm thành viên của lớp. Các thuộc tính truy cập luôn luôn kết thúc với dấu hai chấm (**:**) và có thể xuất hiện nhiều lần và theo thứ tự bất kỳ trong định nghĩa lớp. Mặc định thuộc tính truy cập là **private**.

Định nghĩa lớp chứa các prototype của bốn hàm thành viên sau thuộc tính truy cập **public** là Time(), SetTime(), PrintMilitary() và PrintStandard(). Đó là các hàm thành viên public (public member function) hoặc giao diện (interface) của lớp. Các hàm này sẽ được sử dụng bởi các client (nghĩa là các phần của một chương trình mà là các người dùng) của lớp xử lý dữ liệu của lớp. Có thể nhận thấy trong định nghĩa lớp Time, hàm thành viên Time() có cùng tên với tên lớp Time, nó được gọi là hàm xây dựng (constructor function) của lớp Time.

Một constructor là một hàm thành viên đặc biệt mà khởi động các thành viên dữ liệu của một đối tượng của lớp. Một constructor của lớp được gọi tự động khi đối tượng của lớp đó được tạo.

Thông thường, các thành viên dữ liệu được liệt kê trong phần **private** của một lớp, còn các hàm thành viên được liệt kê trong phần **public**. Nhưng có thể có các hàm thành viên **private** và thành viên dữ liệu **public**.

Khi lớp được định nghĩa, nó có thể sử dụng như một kiểu trong phần khai báo như sau:

Time Sunset, // Đối tượng của lớp Time

ArrayTimes[5], // Mảng các đối tượng của lớp Time

*PTime, // Con trỏ trỏ đến một đối tượng của lớp Time

&DinnerTime = Sunset; // Tham chiếu đến một đối tượng của lớp Time

Ví dụ 3.2: Xây dựng lại lớp Time ở ví dụ 3.1

```
1: #include <iostream.h>
2:
3: class Time
4: {
5:  public:
6:  Time();  //Constructor
7:  void SetTime(int, int, int); //Thiết lập Hour, Minute va Second
8:  void PrintMilitary(); //In thời gian dưới dạng giờ quân đội
9:  void PrintStandard(); //In thời gian dưới dạng chuẩn
10: private:
11: int Hour; // 0 - 23
12: int Minute; // 0 - 59
13: int Second; // 0 - 59
14: };
15:
16: //Constructor khởi tạo mỗi thành viên DL với giá trị zero
17: //Bảo đảm tất cả các đối tượng bắt đầu ở một t.thái thích hợp
18: Time::Time()
19: {
20: Hour = Minute = Second = 0;
21: }
22:
23: //Thiết lập một giá trị Time mới sử dụng giờ quânđội
24: //Thực hiện việc kiểm tra tính hợp lệ trên các giá trị dữ liệu
25: //Thiết lập các giá trị không hợp lệ thành zero
26: void Time::SetTime(int H, int M, int S)
27: {
28:    Hour = (H >= 0 && H < 24) ? H : 0;
29:    Minute = (M >= 0 && M < 60) ? M : 0;
30:    Second = (S >= 0 && S < 60) ? S : 0;
31: }
32:
33: //In thời gian dưới dạng giờ quân đội
34: void Time::PrintMilitary()
35: {
36:    cout << (Hour < 10 ? "0" : "") << Hour << ":"
37:         << (Minute < 10 ? "0" : "") << Minute << ":"
38:         << (Second < 10 ? "0" : "") << Second;
39: }
40:
41: //In thời gian dưới dạng chuẩn
42: void Time::PrintStandard()
```

```
43: {
44:    cout << ((Hour == 0 || Hour == 12) ? 12 : Hour % 12)
44:         << ":" << (Minute < 10 ? "0" : "") << Minute
45:         << ":" << (Second < 10 ? "0" : "") << Second
46:         << (Hour < 12 ? " AM" : " PM");
48: }
49:
50: int main()
51: {
52:    Time T; //Đối tượng T của lớp Time
53:
54:    cout << "The initial military time is ";
55:    T.PrintMilitary();
56:    cout << endl << "The initial standard time is ";
57:    T.PrintStandard();
58:
59:    T.SetTime(13, 27, 6);
60:    cout << endl << endl << "Military time after SetTime is ";
61:    T.PrintMilitary();
62:    cout << endl << "Standard time after SetTime is ";
63:    T.PrintStandard();
64:
65:    T.SetTime(99, 99, 99); //Thử thiết lập giá trị không hợp lệ
66:    cout << endl << endl << "After attempting invalid settings:"
67:         << endl << "Military time: ";
68:    T.PrintMilitary();
69:    cout << endl << "Standard time: ";
70:    T.PrintStandard();
71:    cout << endl;
72:    return 0;
73: }
```

Chúng ta chạy ví dụ 3.2, kết quả ở hình 3.2

```
The initial military time is 00:00:00
The initial standard time is 12:00:00 AM

Military time after SetTime is 13:27:06
Standard time after SetTime is 1:27:06 PM

After attempting invalid settings:
Military time: 00:00:00
Standard time: 12:00:00 AM
```

Hình 3.2: Kết quả của ví dụ 3.2

Trong ví dụ 3.2, chương trình thuyết minh một đối tượng của lớp *Time* gọi là *T* (dòng 52). Khi đó constructor của lớp *Time* tự động gọi và rõ ràng khởi tạo mỗi thành viên dữ liệu **private** là zero. Sau đó thời gian được in dưới dạng giờ quân đội và dạng chuẩn để xác nhận các thành viên này được khởi tạo thích hợp (dòng 54 đến 57). Kế tới thời gian được thiết lập bằng cách sử dụng hàm thành viên *SetTime*() (dòng 59) và thời gian lại được in ở hai dạng (dòng 60 đến 63). Cuối cùng hàm thành viên *SetTime*() (dòng 65) thử thiết lập các thành viên dữ liệu với các giá trị không hợp lệ, và thời gian lại được in ở hai dạng (dòng 66 đến 70).

Chúng ta nhận thấy rằng, tất cả các thành viên dữ liệu của một lớp không thể khởi tạo tại nơi mà chúng được khai báo trong thân lớp. Các thành viên dữ liệu này phải được khởi tạo bởi constructor của lớp hay chúng có thể gán giá trị bởi các hàm thiết lập.

Khi một lớp được định nghĩa và các hàm thành viên của nó được khai báo, các hàm thành viên này phải được định nghĩa. Mỗi hàm thành viên của lớp có thể được định nghĩa trực tiếp trong thân lớp (hiển nhiên bao gồm prototype hàm của lớp), hoặc hàm thành viên có thể được định nghĩa sau thân lớp. Khi một hàm thành viên được định nghĩa sau định nghĩa lớp tương ứng, tên hàm được đặt trước bởi tên lớp và toán tử định phạm vi (**::**). Chẳng hạn như ở ví dụ 3.2 gồm các dòng 18, 26, 34 và 42. Bởi vì các lớp khác nhau có thể có các tên thành viên giống nhau, toán tử định phạm vi "ràng buộc" tên thành viên tới tên lớp để nhận dạng các hàm thành viên của một lớp.

Mặc dù một hàm thành viên khai báo trong định nghĩa một lớp có thể định nghĩa bên ngoài định nghĩa lớp này, hàm thành viên đó vẫn còn bên trong phạm vi của lớp, nghĩa là tên của nó chỉ được biết tới các thành viên khác của lớp ngoại trừ tham chiếu thông qua một đối tượng của lớp, một tham chiếu tới một đối tượng của lớp, hoặc một con trỏ trỏ tới một đối tượng của lớp.

Nếu một hàm thành viên được định nghĩa trong định nghĩa một lớp, hàm thành viên này chính là hàm inline. Các hàm thành viên định nghĩa bên ngoài định nghĩa một lớp có thể là hàm inline bằng cách sử dụng từ khóa **inline**.

Hàm thành viên cùng tên với tên lớp nhưng đặt trước là một ký tự ngã (~) được gọi là destructor của lớp này. Hàm destructor làm "công việc nội trợ kết thúc" trên mỗi đối tượng của lớp trước khi vùng nhờ cho đối tượng được phục hồi bởi hệ thống.

Ví dụ 3.3: Lấy lại ví dụ 3.2 nhưng hai hàm *PrintMilitary*() và *PrintStandard*() là các hàm inline.

```
1: #include <iostream.h>
2:
3: class Time
4: {
5: public:
6: Time(); ; //Constructor
7: void SetTime(int, int, int); //Thiết lập Hour, Minute va
Second
8: void PrintMilitary() // In thời gian dưới dạng giờ quânđội
9: {
10: cout << (Hour < 10 ? "0" : "") << Hour << ":"
11:          << (Minute < 10 ? "0" : "") << Minute << ":"
12:          << (Second < 10 ? "0" : "") << Second;
13: }
14: void PrintStandard(); // In thời gian dưới dạng chuẩn
15: private:
16: int Hour; // 0 - 23
17: int Minute; // 0 - 59
18: int Second; // 0 - 59
19: };
20: //Constructor khởi tạo mỗi thành viên dữ liệu với giá trị zero
21: //Bảo đảm t.cả các đối tượng bắt đầu ở một trạng thái thích hợp
22: Time::Time()
23: {
24: Hour = Minute = Second = 0;
25: }
26:
27: #9; //Thiết lập một giá trị Time mới sử dụng giờ quân đội
28: #9; //T.hiện việc k.tra tính hợp lệ trên các giá trị DL
29: #9; //Thiết lập các giá trị không hợp lệ thành zero
30: void Time::SetTime(int H, int M, int S)
31: {
32: Hour = (H >= 0 && H < 24) ? H : 0;
33: Minute = (M >= 0 && M < 60) ? M : 0;
34: Second = (S >= 0 && S < 60) ? S : 0;
```

```
35: }
36:
37: #9; //In thời gian dưới dạng chuẩn
38: inline void Time::PrintStandard()
39: {
40: cout << ((Hour == 0 || Hour == 12) ? 12 : Hour % 12)
41:       << ":" << (Minute < 10 ? "0" : "") << Minute
42:       << ":" << (Second < 10 ? "0" : "") << Second
43: << (Hour < 12 ? " AM" : " PM");
44: }
45:
46: int main()
47: {
48: Time T;
49:
50: cout << "The initial military time is ";
51: T.PrintMilitary();
52: cout << endl << "The initial standard time is ";
53: T.PrintStandard();
54:
55: T.SetTime(13, 27, 6);
56: cout << endl << endl << "Military time after SetTime is ";
57: T.PrintMilitary();
58: cout << endl << "Standard time after SetTime is ";
59: T.PrintStandard();
60:
61: T.SetTime(99, 99, 99); //Thử thiết lập giá trị không hợp lệ
62: cout << endl << endl << "After attempting invalid settings:"
63:       << endl << "Military time: ";
64: T.PrintMilitary();
65: cout << endl << "Standard time: ";
66: T.PrintStandard();
67: cout << endl;
68: return 0;
69: }
```

Chúng ta chạy ví dụ 3.3, kết quả ở hình 3.3

```
The initial military time is 00:00:00
The initial standard time is 12:00:00 AM

Military time after SetTime is 13:27:06
Standard time after SetTime is 1:27:06 PM

After attempting invalid settings:
Military time: 00:00:00
Standard time: 12:00:00 AM
```

Hình 3.3: Kết quả của ví dụ 3.3

IV. PHẠM VI LỚP VÀ TRUY CẬP CÁC THÀNH VIÊN LỚP

Các thành viên dữ liệu của một lớp (các biến khai báo trong định nghĩa lớp) và các hàm thành viên (các hàm khai báo trong định nghĩa lớp) thuộc vào phạm vi của lớp.

Trong một phạm vi lớp, các thành viên của lớp được truy cập ngay lập tức bởi tất cả các hàm thành viên của lớp đó và có thể được tham chiếu một cách dễ dàng bởi tên. Bên ngoài một phạm vi lớp, các thành viên của lớp được tham chiếu thông qua hoặc một tên đối tượng, một tham chiếu đến một đối tượng, hoặc một con trỏ tới đối tượng.

Các hàm thành viên của lớp có thể được đa năng hóa (overload), nhưng chỉ bởi các hàm thành viên khác của lớp. Để đa năng hóa một hàm thành viên, đơn giản cung cấp trong định nghĩa lớp một prototype cho mỗi phiên bản của hàm đa năng hóa, và cung cấp một định nghĩa hàm riêng biệt cho mỗi phiên bản của hàm.

Các hàm thành viên có phạm vi hàm trong một lớp – các biến định nghĩa trong một hàm thành viên chỉ được biết tới hàm đó. Nếu một hàm thành viên định nghĩa một biến cùng tên với tên một biến trong phạm vi lớp, biến phạm vi lớp được dấu bởi biến phạm vi hàm bên trong phạm vi hàm. Như thế một biến bị dấu có thể được truy cập thông qua toán tử định phạm vi.

Các toán tử được sử dụng để truy cập các thành viên của lớp được đồng nhất với các toán tử sử dụng để truy cập các thành viên của cấu trúc. Toán tử lựa chọn thành viên dấu chấm (**.**) được kết hợp với một tên của đối tượng hay với một tham chiếu tới một đối tượng để truy cập các thành viên của đối tượng. Toán tử lựa chọn thành viên mũi tên (**->**)được kết hợp với một con trỏ trỏ tới một truy cập để truy cập các thành viên của đối tượng.

Ví dụ 3.4: Chương trình sau minh họa việc truy cập các thành viên của một lớp với các toán tử lựa chọn thành viên.

```
1: #include <iostream.h>
2:
3: class Count
4: {
5:     public:
6:     int X;
7:     void Print()
8:     {
9:         cout << X << endl;
10:     }
11: };
12:
13: int main()
14: {
15:     Count Counter, //Tạo đối tượng Counter
16:     *CounterPtr = &Counter, //Con trỏ trỏ tới Counter
17:     &CounterRef = Counter; //Tham chiếu tới Counter
18:
19:     cout << "Assign7 to X and Print using the object's name: ";
20:     Counter.X = 7; //Gán 7 cho thành viên dữ liệu X
21:     Counter.Print(); //Gọi hàm thành viên Print
22:
23:     cout << "Assign 8 to X and Print using a reference: ";
24:     CounterRef.X = 8; //Gán 8 cho thành viên dữ liệu X
25:     CounterRef.Print(); //Gọi hàm thành viên Print
26:
27:     cout << "Assign 10 to X and Print using a pointer: ";
28:     CounterPtr->X = 10; // Gán 10 cho thành viên dữ liệu X
29:     CounterPtr->Print(); //Gọi hàm thành viên Print
30:     return 0;
31: }
```

Chúng ta chạy ví dụ 3.4, kết quả ở hình 3.4

```
Assign 7 to X and Print using the object's name: 7
Assign 8 to X and Print using a reference: 8
Assign 10 to X and Print using a pointer: 10
```

Hình 3.4: Kết quả của ví dụ 3.4

V. ĐIỀU KHIỂN TRUY CẬP TỚI CÁC THÀNH VIÊN

Các thuộc tính truy cập **public** và **private** (và **protected** chúng ta sẽ xem xét sau) được sử dụng để điều khiển truy cập tới các thành viên dữ liệu và các hàm thành viên của lớp. Chế độ truy cập mặc định đối với lớp là **private** vì thế tất cả các thành viên sau phần header của lớp và trước nhãn đầu tiên là **private**. Sau mỗi nhãn, chế độ mà được kéo theo bởi nhãn đó áp dụng cho đến khi gặp nhãn kế tiếp hoặc cho đến khi gặp dấu móc phải (}) của phần định nghĩa lớp. Các nhãn **public**, **private** và **protected** có thể được lặp lại nhưng cách dùng như vậy thì hiếm có và có thể gây khó hiểu.

Các thành viên **private** chỉ có thể được truy cập bởi các hàm thành viên (và các hàm friend) của lớp đó. Các thành viên **public** của lớp có thể được truy cập bởi bất kỳ hàm nào trong chương trình.

Mục đích chính của các thành viên **public** là để biểu thị cho client của lớp một cái nhìn của các dịch vụ (services) mà lớp cung cấp. Tập hợp này của các dịch vụ hình thành giao diện public của lớp. Các client của lớp không cần quan tâm làm thế nào lớp hoàn thành các thao tác của nó. Các thành viên **private** của lớp cũng như các định nghĩa của các hàm thành viên **public** của nó thì không phải có thể truy cập tới client của một lớp. Các thành phần này hình thành sự thi hành của lớp.

Ví dụ 3.5: Chương trình sau cho thấy rằng các thành viên **private** chỉ có thể truy cập thông qua giao diện public sử dụng các hàm thành viên **public**.

```
#include <iostream.h>
class MyClass
{
   private:
         int X,Y;
   public:
         void Print();
};
void MyClass::Print()
{
   cout <<X<<Y<<endl;
}
int main()
{
   MyClass M;
   M.X = 3;
   M.Y = 4;
   M.Print();
   return 0;
}
```

Khi chúng ta biên dịch chương trình này, compiler phát sinh ra hai lỗi tại hai dòng 20 và 21 như sau:

```
Compiling CT3_5.CPP:
Error CT3_5.CPP 20: 'MyClass::X' is not accessible
Error CT3_5.CPP 21: 'MyClass::Y' is not accessible
```

Hình 3.5: Thông báo lỗi của ví dụ 3.5

Thuộc tính truy cập mặc định đối với các thành viên của lớp là **private**. Thuộc tính truy cập các thành viên của một lớp có thể được thiết lập rõ ràng là **public**, **protected** hoặc **private**. Thuộc tính truy cập mặc

định đối với các thành viên của **struct** là **public**. Thuộc tính truy cập các thành viên của một **struct** cũng có thể được thiết lập rõ ràng là **public**, **protected** hoặc **private**.

Truy cập đến một dữ liệu **private** cần phải được điều khiển cẩn thận bởi việc sử dụng của các hàm thành viên, gọi là các hàm truy cập (access functions).

VI. CÁC HÀM TRUY CẬP VÀ CÁC HÀM TIỆN ÍCH

Không phải tất cả các hàm thành viên đều là **public** để phục vụ như bộ phận giao diện của một lớp. Một vài hàm còn lại là **private** và phục vụ như các hàm tiện ích (utility functions) cho các hàm khác của lớp.

Các hàm truy cập có thể đọc hay hiển thị dữ liệu. Sử dụng các hàm truy cập để kiểm tra tính đúng hoặc sai của các điều kiện – các hàm như thế thường được gọi là các hàm khẳng định (predicate functions). Một ví dụ của hàm khẳng định là một hàm *IsEmpty*() của lớp container - một lớp có khả năng giữ nhiều đối tượng - giống như một danh sách liên kết, một stack hay một hàng đợi. Một chương trình sẽ kiểm tra hàm *IsEmpty*() trước khi thử đọc mục khác từ đối tượng container.

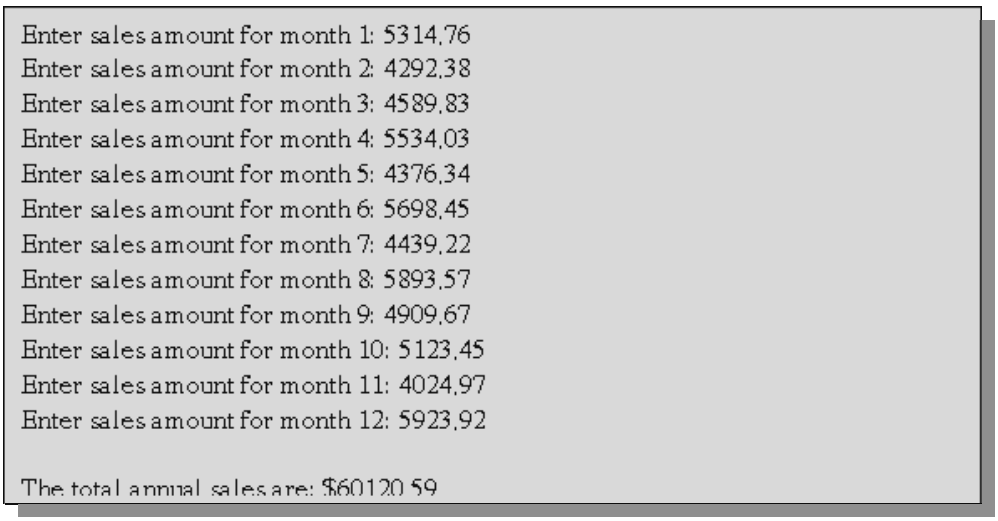
Một hàm tiện ích không là một phần của một giao diện của lớp. Hơn nữa nó là một hàm thành viên **private** mà hỗ trợ các thao tác của các hàm thành viên **public**. Các hàm tiện ích không dự định được sử dụng bởi các client của lớp.

Ví dụ 3.6: Minh họa cho các hàm tiện ích.

```
#include <iostream.h>
#include <iomanip.h>

class SalesPerson
{
public:
SalesPerson();              //constructor
void SetSales(int, double);//Ng.dùng cung cấp các hình của
#9; #9; //những hàng bán của một tháng
void PrintAnnualSales();

private:
double Sales[12]; //12 hình của những hàng bán hằng tháng
double TotalAnnualSales(); //Hàm tiện ích
};

//Hàm constructor khởi tạo mảng
SalesPerson::SalesPerson()
{
for (int I = 0; I < 12; I++)
Sales[I] = 0.0;
}

//Hàm th.lập một trong 12 hình của những hàng bán hằng tháng
void SalesPerson::SetSales(int Month, double Amount)
{
if (Month >= 1 && Month <= 12 && Amount > 0)
Sales[Month - 1] = Amount;
else
cout << "Invalid month or sales figure" << endl;
}

//Hàm tiện íchdể tính tổng hàng bán hằng năm
double SalesPerson::TotalAnnualSales()
{
double Total = 0.0;

```

```
38: for (int I = 0; I < 12; I++)
39: Total += Sales[I];
40: return Total;
41: }
42:
43: //In tổng hàng bán hằng năm
44: void SalesPerson::PrintAnnualSales()
45: {
46: cout << setprecision(2)
47:       << setiosflags(ios::fixed | ios::showpoint)
48:       << endl << "The total annual sales are: $"
49:       << TotalAnnualSales() << endl;
50: }
51:
52: int main()
53: {
54: SalesPerson S;
55: double salesFigure;
56:
57: for (int I = 1; I <= 12; I++)
58: {
59: cout << "Enter sales amount for month "<< I << ": ";
60: cin >> salesFigure;
61: S.SetSales(I, salesFigure);
62: }
63: S.PrintAnnualSales();
64: return 0;
65: }
```

Chúng ta chạy ví dụ 3.6 , kết quả ở hình 3.6



Hình 3.6: Kết quả của ví dụ 3.6

VII. KHỞI ĐỘNG CÁC ĐỐI TƯỢNG CỦA LỚP : CONSTRUCTOR

Khi một đối tượng được tạo, các thành viên của nó có thể được khởi tạo bởi một hàm constructor. Một constructor là một hàm thành viên với tên giống như tên của lớp. Lập trình viên cung cấp constructor mà được gọi tự động mỗi khi đối tượng của lớp đó được tạo. Các thành viên dữ liệu của một lớp không thể được khởi tạo trong định nghĩa của lớp. Hơn nữa, các thành viên dữ liệu phải được khởi động hoặc trong một constructor của lớp hoặc các giá trị của chúng có thể được thiết lập sau sau khi đối tượng được tạo. Các constructor không thể mô tả các kiểu trả về hoặc các giá trị trả về. Các constructor có thể được đa năng hóa để cung cấp sự đa dạng để khởi tạo các đối tượng của lớp.

Constructor có thể chứa các tham số mặc định. Bằng cách cung cấp các tham số mặc định cho constructor, ngay cả nếu không có các giá trị nào được cung cấp trong một constructor thì đối tượng vẫn được bảo đảm để trong một trạng thái phù hợp vì các tham số mặc định. Một constructor của lập trình viên cung cấp mà hoặc tất cả các tham số của nó có giá trị mặc định hoặc không có tham số nào được gọi là constructor mặc định (default constructor). Chỉ có thể có một constructor mặc định cho mỗi lớp.

Ví dụ 3.7: Constructor với các tham số mặc định

```
#include <iostream.H>
class Time
{
  public:
        Time(int = 0, int = 0, int = 0); //Constructor mac dinh
        void SetTime(int, int, int);
        void PrintMilitary();
        void PrintStandard();
  private:
        int Hour;
        int Minute;
        int Second;
};
//Ham constructor de khoi dong du lieu private
//Cac gia tri mac dinh la 0
Time::Time(int Hr, int Min, int Sec)
{
  SetTime(Hr, Min, Sec);
}
//Thiet lap cac gia tri cua Hour, Minute va Second
//Gia tri khong hop le duoc thiet lap la 0
void Time::SetTime(int H, int M, int S)
{
  Hour = (H >= 0 && H < 24) ? H : 0;
  Minute = (M >= 0 && M < 60) ? M : 0;
  Second = (S >= 0 && S < 60) ? S : 0;
}
//Hien thi thoi gian theo dang gio quan doi: HH:MM:SS
void Time::PrintMilitary()
{
  cout << (Hour < 10 ? "0" : "") << Hour << ":"
        << (Minute < 10 ? "0" : "") << Minute << ":"
        << (Second < 10 ? "0" : "") << Second;
}
//Hien thi thoi gian theo dang chuan: HH:MM:SS AM (hoac PM)
void Time::PrintStandard()
{
  cout << ((Hour == 0 || Hour == 12) ? 12 : Hour % 12)
        << ":" << (Minute < 10 ? "0" : "") << Minute
        << ":" << (Second < 10 ? "0" : "") << Second
        << (Hour < 12 ? " AM" : " PM");
}
```

```
int main()
{
  Time T1,T2(2),T3(21,34),T4(12,25,42),T5(27,74,99);

  cout << "Constructed with:" << endl   << "all arguments defaulted:" << endl << "   ";
  T1.PrintMilitary();     cout << endl << "   ";
  T1.PrintStandard();
  cout << endl << "Hour specified; Minute and Second defaulted:"  << endl << "   ";
  T2.PrintMilitary();     cout << endl << "   ";
  T2.PrintStandard();
  cout << endl << "Hour and Minute specified; Second defaulted:"  << endl << "   ";
  T3.PrintMilitary();     cout << endl << "   ";
  T3.PrintStandard();     cout << endl << "Hour, Minute, and Second specified:"<<endl<<"   ";
  T4.PrintMilitary();     cout << endl << "   ";
  T4.PrintStandard();     cout << endl << "all invalid values specified:"   << endl << "   ";
  T5.PrintMilitary();     cout << endl << "   ";
  T5.PrintStandard();     cout << endl;
  return 0;
}
```

Chương trình ở ví dụ 3.7 khởi tạo năm đối tượng của lớp *Time* (ở dòng 52). Đối tượng *T1* với ba tham số lấy giá trị mặc định, đối tượng *T2* với một tham số được mô tả, đối tượng *T3* với hai tham số được mô tả, đối tượng *T4* với ba tham số được mô tả và đối tượng *T5* với các tham số có giá trị không hợp lệ.

Chúng ta chạy ví dụ 3.7, kết quả ở hình 3.7

```
Constructed with:
all arguments defaulted:
  00:00:00
  12:00:00 AM
Hour specified; Minute and Second defaulted:
  02:00:00
  2:00:00 AM
Hour and Minute specified; Second defaulted:
  21:34:00
  9:34:00 PM
Hour, Minute, and Second specified:
  12:25:42
  12:25:42 PM
all invalid values specified:
  00:00:00
  12:00:00 AM
```

Hình 3.7: Kết quả của ví dụ 3.7

Nếu không có constructor nào được định nghĩa trong một lớp thì trình biên dịch tạo một constructor mặc định. Constructor này không thực hiện bất kỳ sự khởi tạo nào, vì vậy khi đối tượng được tạo, nó không bảo đảm để trong một trạng thái phù hợp.

VIII. SỬ DỤNG DESTRUCTOR

Một destructor là một hàm thành viên đặc biệt của một lớp. Tên của destructor đối với một lớp là ký tự ngã (~) theo sau bởi tên lớp.

Destructor của một lớp được gọi khi đối tượng được hủy bỏ nghĩa là khi sự thực hiện chương trình rời khỏi phạm vi mà trong đó đối tượng của lớp đó được khởi tạo. Destructor không thực sự hủy bỏ đối tượng – nó thực hiện "công việc nội trợ kết thúc" trước khi hệ thống phục hồi không gian bộ nhớ của đối tượng để nó có thể được sử dụng giữ các đối tượng mới.

Một destructor không nhận các tham số và không trả về giá trị. Một lớp chỉ có duy nhất một destructor – đa năng hóa destructor là không cho phép.

Nếu trong một lớp không có định nghĩa một destructor thì trình biên dịch sẽ tạo một destructor mặc định không làm gì cả.

Ví dụ 3.8: Lớp có hàm destructor

```
#include <iostream.h>
class Simple
{private:
   int *X;
 public:
   Simple(); //Constructor
   ~Simple(); //Destructor
   void SetValue(int V);
   int GetValue();
};
Simple::Simple()
{    X = new int; //Cấp phát vùng nhớ cho X
}
Simple::~Simple()
{
    delete X; //Giải phóng vùng nhớ khi đối tượng bị hủy bỏ.
}
void Simple::SetValue(int V)
{
    *X = V;
}
int Simple::GetValue()
{
    return *X;
}
int main()
{
   Simple S;
   int X;
   cout<<"Enter a number:";
   cin>>X;
   S.SetValue(X);
   cout<<"The value of this number:"<<S.GetValue();
   return 0;
}
```

Chúng ta chạy ví dụ 3.8, kết quả ở hình 3.8

```
Enter a number: 12
The value of this number: 12
```

Hình 3.8: Kết quả của ví dụ 3.8

IX. KHI NÀO CÁC CONSTRUTOR VÀ DESTRUCTOR ĐƯỢC GỌI ?

Các constructor và destructor được gọi một cách tự động. Thứ tự các hàm này được gọi phụ thuộc vào thứ tự trong đó sự thực hiện vào và rời khỏi phạm vi mà các đối tượng được khởi tạo. Một cách tổng quát, các destructor được gọi theo thứ tự ngược với thứ tự của các constructor được gọi.

Các constructor được gọi của các đối tượng khai báo trong phạm vi toàn cục trước bất kỳ hàm nào (bao gồm hàm **main()**) trong file mà bắt đầu thực hiện. Các destructor tương ứng được gọi khi hàm **main()** kết thúc hoặc hàm **exit()** được gọi.

Các constructor của các đối tượng cục bộ tự động được gọi khi sự thực hiện đến điểm mà các đối tượng được khai báo. Các destructor tương ứng được gọi khi các đối tượng rời khỏi phạm vi (nghĩa là khối mà trong đó chúng được khai báo). Các constructor và destructor đối với các đối tượng cục bộ tự động được gọi mỗi khi các đối tượng vào và rời khỏi phạm vi.

Các constructor được gọi của các đối tượng cục bộ tĩnh (static) khi sự thực hiện đến điểm mà các đối tượng được khai báo lần đầu tiên. Các destructor tương ứng được gọi khi hàm **main()** kết thúc hoặc hàm **exit()** được gọi.

Ví dụ 3.9: Chương trình sau minh họa thứ tự các constructor và destructor được gọi.

```
#include <iostream.h>
class CreateAndDestroy
{
      public:
            CreateAndDestroy(int);  //Constructor
            ~CreateAndDestroy();    //Destructor
      private:
            int Data;
};
CreateAndDestroy::CreateAndDestroy(int Value)
{
      Data = Value;
      cout << "Object " << Data << "   constructor";
}
CreateAndDestroy::~CreateAndDestroy()
{
      cout << "Object " << Data << "   destructor " << endl;
}
void Create(void);   //Prototype
CreateAndDestroy First(1);  //Doi tuong toan cuc
int main()
{
      cout << "   (global created before main)" << endl;
      CreateAndDestroy Second(2);        //Doi tuong cuc bo
      cout << "   (local automatic in main)" << endl;
      static CreateAndDestroy Third(3);  //Doi tuong cuc bo
      cout << "   (local static in main)" << endl;
      Create();  //Goi ham de tao cac doi tuong
      CreateAndDestroy Fourth(4);        //Doi tuong cuc bo
      cout << "   (local automatic in main)" << endl;
      return 0;
}
//Ham tao cac doi tuong
void Create(void)
{
      CreateAndDestroy Fifth(5);
      cout << "   (local automatic in create)" << endl;
      static CreateAndDestroy Sixth(6);
      cout << "   (local static in create)" << endl;
```

```
        CreateAndDestroy Seventh(7);
        cout << "   (local automatic in create)" << endl;
}
```

Chương trình khai báo *First* ở phạm vi toàn cục. Constructor của nó được gọi khi chương trình bắt đầu thực hiện và destructor của nó được gọi lúc chương trình kết thúc sau tất cả các đối tượng khác được hủy bỏ. Hàm main() khai báo ba đối tượng. Các đối tượng *Second* và *Fourth* là các đối tượng cục bộ tự động và đối tượng *Third* là một đối tượng cục bộ tĩnh. Các constructor của các đối tượng này được gọi khi chương trình thực hiện đến điểm mà mỗi đối tượng được khai báo. Các destructor của các đối tượng *Fourth* và *Second* được gọi theo thứ tự này khi kết thúc của **main()** đạt đến. Vì đối tượng *Third* là tĩnh, nó tồn tại cho đến khi chương trình kết thúc. Destructor của đối tượng *Third* được gọi trước destructor của *First* nhưng sau tất cả các đối tượng khác được hủy bỏ.

Hàm *Create()* khai báo ba đối tượng – *Fifth* và *Seventh* là các đối tượng cục bộ tự động và *Sixth* là một đối tượng cục bộ tĩnh. Các destructor của các đối tượng *Seventh* và Fifth được gọi theo thứ tự này khi kết thúc của *create()* đạt đến. Vì đối tượng *Sixth* là tĩnh, nó tồn tại cho đến khi chương trình kết thúc. Destructor của đối tượng *Sixth* được gọi trước các destructor của *Third* và *First* nhưng sau tất cả các đối tượng khác được hủy bỏ.

Chúng ta chạy ví dụ 3.9, kết quả ở hình 3.9

```
Object 1  constructor  (global created before main)
Object 2  constructor  (local automatic in main)
Object 3  constructor  (local static in main)
Object 5  constructor  (local automatic in create)
Object 6  constructor  (local static in create)
Object 7  constructor  (local automatic in create)
Object 7  destructor
Object 5  destructor
Object 4  constructor  (local automatic in main)
Object 4  destructor
Object 2  destructor
Object 6  destructor
Object 3  destructor
Object 1  destructor
```

Hình 3.9: Kết quả của ví dụ 3.9

X. SỬ DỤNG CÁC THÀNH VIÊN DỮ LIỆU VÀ CÁC HÀM THÀNH VIÊN

Các thành viên dữ liệu **private** chỉ có thể được xử lý bởi các hàm thành viên (hay hàm **friend**) của lớp. Các lớp thường cung cấp các hàm thành viên **public** để cho phép các client của lớp để thiết lập (set) (nghĩa là "ghi") hoặc lấy (get) (nghĩa là "đọc") các giá trị của các thành viên dữ liệu **private**. Các hàm này thường không cần phải được gọi "set" hay "get", nhưng chúng thường đặt tên như vậy. Chẳng hạn, một lớp có thành viên dữ liệu **private** có tên *InterestRate*, hàm thành viên thiết lập giá trị có tên là *SetInterestRate*() và hàm thành viên lấy giá trị có tên là *GetInterestRate*(). Các hàm "Get" cũng thường được gọi là các hàm chất vấn (query functions).

Nếu một thành viên dữ liệu là **public** thì thành viên dữ liệu có thể được đọc hoặc ghi tại bất kỳ hàm nào trong chương trình. Nếu một thành viên dữ liệu là **private**, một hàm "get" public nhất định cho phép các hàm khác để đọc dữ liệu nhưng hàm get có thể điều khiển sự định dạng và hiển thị của dữ liệu. Một hàm "set" public có thể sẽ xem xét cẩn thận bất kỳ cố gắng nào để thay đổi giá trị của thành viên dữ liệu. Điều này sẽ bảo đảm rằng giá trị mới thì tương thích đối với mục dữ liệu. Chẳng hạn, một sự cố gắng thiết lập ngày của tháng là 37 sẽ bị loại trừ.

Các lợi ích của sự toàn vẹn dữ liệu thì không tự động đơn giản bởi vì các thành viên dữ liệu được tạo là **private** – lập trình viên phải cung cấp sự kiểm tra hợp lệ. Tuy nhiên C++ cung cấp một khung làm việc trong đó các lập trình viên có thể thiết kế các chương trình tốt hơn.

Client của lớp phải được thông báo khi một sự cố gắng được tạo ra để gán một giá trị không hợp lệ cho một thành viên dữ liệu. Chính vì lý do này, các hàm "set" của lớp thường được viết trả về các giá trị cho biết rằng một sự cố gắng đã tạo ra để gán một dữ liệu không hợp lệ cho một đối tượng của lớp. Điều này cho phép các client của lớp kiểm tra các giá trị trả về để xác định nếu đối tượng mà chúng thao tác là một đối tượng hợp lệ và để bắt giữ hoạt động thích hợp nếu đối tượng mà chúng thao tác thì không phải hợp lệ.

Ví dụ 3.10: Chương trình mở rộng lớp *Time* ở ví dụ 3.2 bao gồm hàm get và set đối với các thành viên dữ liệu private là *hour*, *minute* và *second*.

```
1: #include <iostream.h>
2:
3: class Time
4: {
5: public:
6: Time(int = 0, int = 0, int = 0); //Constructor
7: //Các hàm set
8: void SetTime(int, int, int); //Thiết lập Hour, Minute, Second
9: void SetHour(int); //Thiết lập Hour
10: void SetMinute(int); //Thiết lập Minute
11: void SetSecond(int); //Thiết lập Second
12: //Các hàm get
13: int GetHour();    //Trả về Hour
14: int GetMinute();    //Trả về Minute
15: int GetSecond();    //Trả về Second
16:
17: void PrintMilitary(); //Xuất thời gian theo dạng giờ quânđội
18: void PrintStandard(); //Xuất thời gian theo dạng chuẩn
19:
20: private:
21: int Hour; //0 - 23
22: int Minute; //0 - 59
23: int Second; //0 – 59
24: };
25:
26: //Constructor khởiđộng dữ liệu private
27: //Gọi hàm thành viên SetTime() để thiết lập các biến
24: //Các giá trị mặc định là 0
25: Time::Time(int Hr, int Min, int Sec)
26: {
27: SetTime(Hr, Min, Sec);
28: }
29:
30: //Thiết lập các giá trị của Hour, Minute, và Second
31: void Time::SetTime(int H, int M, int S)
32: {
33: Hour = (H >= 0 && H < 24) ? H : 0;
34: Minute = (M >= 0 && M < 60) ? M : 0;
35: Second = (S >= 0 && S < 60) ? S : 0;
36: }
37:
38: //Thiết lập giá trị của Hour
39: void Time::SetHour(int H)
40: {
41: Hour = (H >= 0 && H < 24) ? H : 0;
42: }
```

```
43:
44: //Thiết lập giá trị của Minute
45: void Time::SetMinute(int M)
46: {
47: Minute = (M >= 0 && M < 60) ? M : 0;
48: }
49:
50: //Thiết lập giá trị của Second
51: void Time::SetSecond(int S)
52: {
53: Second = (S >= 0 && S < 60) ? S : 0;
54: }
55:
56: //Lấy giá trị của Hour
57: int Time::GetHour()
58: {
59: return Hour;
60: }
61:
62: //Lấy giá trị của Minute
63: int Time::GetMinute()
64: {
65: return Minute;
66: }
67:
68: //Lấy giá trị của Second
69: int Time::GetSecond()
70: {
71: return Second;
72: }
73:
74: //Hiển thị thời gian dạng giờ quânđội: HH:MM:SS
75: void Time::PrintMilitary()
76: {
77: cout << (Hour < 10 ? "0" : "") << Hour << ":"
78:       << (Minute < 10 ? "0" : "") << Minute << ":"
79:       << (Second < 10 ? "0" : "") << Second;
80: }
81:
83: //Hiển thị thời gian dạng chuẩn: HH:MM:SS AM (hay PM)
84: void Time::PrintStandard()
85: {
86: cout << ((Hour == 0 || Hour == 12) ? 12 : Hour % 12) << ":"
87:       << (Minute < 10 ? "0" : "") << Minute << ":"
88:       << (Second < 10 ? "0" : "") << Second
89:       << (Hour < 12 ? " AM" : " PM");
90: }
91:
92: void IncrementMinutes(Time &, const int); //prototype
93:
94: int main()
95: {
96: Time T;
97:
99: T.SetHour(17);
100: T.SetMinute(34);
101: T.SetSecond(25);
102 cout << "Result of setting all valid values:" << endl
```

```
103:       << " Hour: " << T.GetHour()
104:       << " Minute: " << T.GetMinute()
105:       << " Second: " << T.GetSecond() << endl << endl;
106: T.SetHour(234); //Hour không hợp lệđược thiết lập bằng 0
107: T.SetMinute(43);
108: T.SetSecond(6373); //Second không hợp lệđược thiết lập bằng 0
109: cout << "Result of attempting to set invalid Hour and"
110:       << " Second:" << endl << " Hour: " << T.GetHour()
111:       << " Minute: " << T.GetMinute()
112:       << " Second: " << T.GetSecond() << endl << endl;
113: T.SetTime(11, 58, 0);
114: IncrementMinutes(T, 3);
115: return 0;
116: }
117:
118: void IncrementMinutes(Time &TT, const int Count)
119: {
120: cout << "Incrementing Minute " << Count
121:       << " times:" << endl << "Start time: ";
122: TT.PrintStandard();
123: for (int I = 1; I <= Count; I++)
124: {
125: TT.SetMinute((TT.GetMinute() + 1) % 60);
126: if (TT.GetMinute() == 0)
127: TT.SetHour((TT.GetHour() + 1) % 24);
128: cout << endl << "Minute + 1: ";
129: TT.PrintStandard();
130: }
131: cout << endl;
132: }
```

Trong ví dụ trên chúng ta có hàm *IncrementMinutes*() là hàm dùng để tăng *Minite*. Đây là hàm không thành viên mà sử dụng các hàm thành viên get và set để tăng thành viên *Minite*.

Chúng ta chạy ví dụ .10, kết quả ở hình 3.10

```
Result of setting all valid values:
 Hour: 17  Minute: 34  Second: 25

Result of attempting to set invalid Hour and Second:
 Hour: 0  Minute: 43  Second: 0

Incrementing Minute 3 times:
Start time: 11:58:00 AM
Minute + 1: 11:59:00 AM
Minute + 1: 12:00:00 PM
Minute + 1: 12:01:00 PM
```

Hình 3.10: Kết quả của ví dụ 3.10

XI. TRẢ VỀ MỘT THAM CHIẾU TỚI MỘT THÀNH VIÊN DỮ LIỆU PRIVATE

Một tham chiếu tới một đối tượng là một bí danh của chính đối tượng đó và do đó có thể được sử dụng ở vế trái của phép gán. Trong khung cảnh đó, tham chiếu tạo một lvalue được chấp nhận hoàn toàn mà có thể nhận một giá trị. Một cách để sử dụng khả năng này (thật không may!) là có một hàm thành viên **public** của lớp trả về một tham chiếu không **const** tới một thành viên dữ liệu **private** của lớp đó.

Ví dụ 3.11: Chương trình sau sử dụng một phiên bản đơn giản của lớp *Time* để minh họa trả về một tham chiếu tới một dữ liệu **private**.

```
1: #include <iostream.h>
2:
3: class Time
4: {
5: public:
6: Time(int = 0, int = 0, int = 0);
7: void SetTime(int, int, int);
8: int GetHour();
9: int &BadSetHour(int); //Nguy hiểm trả về tham chiếu !!!
10: private:
11: int Hour;
12: int Minute;
13: int Second;
14: };
15:
16: //Constructor khởiđộng dữ liệu private
17: //Gọi hàm thành viên SetTime()để thiết lập các biến
18: //Các giá trị mặcđịnh là 0
19: Time::Time(int Hr, int Min, int Sec)
20: {
21: SetTime(Hr, Min, Sec);
22: }
23: //Thiết lập các giá trị của Hour, Minute, và Second
24: void Time::SetTime(int H, int M, int S)
25: {
26: Hour = (H >= 0 && H < 24) ? H : 0;
27: Minute = (M >= 0 && M < 60) ? M : 0;
28: Second = (S >= 0 && S < 60) ? S : 0;
29: }
30:
31: //Lấy giá trị của Hour
32: int Time::GetHour()
33: {
34: return Hour;
35: }
36:
37: //KHÔNG NÊN LẬP TRÌNH THEO KIỂU NÀY !!!
38: //Trả về một tham chiếu tới một thành viên dữ liệu private
39: int &Time::BadSetHour(int HH)
40: {
41: Hour = (HH >= 0 && HH < 24) ? HH : 0;
42: return Hour; //Nguy hiểm trả về tham chiếu !!!
43: }
44:
45: int main()
46: {
47: Time T;
48: int &HourRef = T.BadSetHour(20);
49:
50: cout << "Hour before modification: " << HourRef << endl;
51: HourRef = 30; //Thayđổi với giá trị không hợp lệ
52: cout << "Hour after modification: " << T.GetHour() << endl;
53: // Nguy hiểm: Hàm trả về một tham chiếu
54: //có thểđược sử dụng như một lvalue
55: T.BadSetHour(12) = 74;
56: cout << endl << "*********************************" << endl
```

```
57:       << "BAD PROGRAMMING PRACTICE!!!!!!!!!" << endl
58:       << "BadSetHour as an lvalue, Hour: "
59:       << T.GetHour()
60:       << endl << "*********************************" << endl;
61: return 0;
62: }
```

Trong chương trình hàm *BadSetHour*() trả về một tham chiếu tới thành viên dữ liệu *Hour*.

Chúng ta chạy ví dụ 3.11, kết quả ở hình 3.11

```
Hour before modification: 20
Hour after modification: 30

*********************************
BAD PROGRAMMING PRACTICE!!!!!!!!!
BadSetHour as an lvalue, Hour: 74
*********************************
```

Hình 3.11: Kết quả của ví dụ 3.11

XII. PHÉP GÁN BỞI TOÁN TỬ SAO CHÉP THÀNH VIÊN MẶC ĐỊNH

Toán tử gán (=) được sử dụng để gán một đối tượng cho một đối tượng khác của cùng một kiểu. Toán tử gán như thế bình thường được thực hiện bởi toán tử sao chép thành viên (Memberwise copy) – Mỗi thành viên của một đối tượng được sao chép riêng rẽ tới cùng thành viên ở đối tượng khác (Chú ý rằng sao chép thành viên có thể phát sinh các vấn đề nghiêm trọng khi sử dụng với một lớp mà thành viên dữ liệu chứa vùng nhớ cấp phát động).

Các đối tượng có thể được truyền cho các tham số của hàm và có thể được trả về từ các hàm. Như thế việc truyền và trả về được thực hiện theo truyền giá trị – một sao chép của đối tượng được truyền hay trả về.:

Ví dụ 3.12: Chương trình sau minh họa toán tử sao chép thành viên mặc định

```
2: #include <iostream.h>
3: //Lớp Date đơn giản
4: class Date
5: {
6: public:
7: Date(int = 1, int = 1, int = 1990); //Constructor mặc định
8: void Print();
9: private:
10: int Month;
11: int Day;
12: int Year;
13: };
14:
15: //Constructor Date đơn giản với việc không kiểm tra miền
16: Date::Date(int m, int d, int y)
17: {
18: Month = m;
19: Day = d;
20: Year = y;
21: }
22:
23: //In Date theo dạng mm-dd-yyyy
24: void Date::Print()
25: {
26: cout << Month << '-' << Day << '-' << Year;
27: }
```

```
28:
29: int main()
30: {
31: Date Date1(7, 4, 1993), Date2; //Date2 mặc định là 1/1/90
32: cout << "Date1 = ";
33: Date1.Print();
34: cout << endl << "Date2 = ";
35: Date2.Print();
36: Date2 = Date1; //Gán bởi toán tử sao chép thành viên mặc định
37: cout << endl << endl
38:       << "After default memberwise copy, Date2 = ";
39: Date2.Print();
40: cout << endl;
41: return 0;
42: }
```

Chúng ta chạy ví dụ 3.12, kết quả ở hình 3.12

```
Date1 = 7-4-1993
Date2 = 1-1-1990

After default memberwise copy, Date2 = 7-4-1993
```

Hình 3.12: Kết quả của ví dụ 3.12

XIII. CÁC ĐỐI TƯỢNG HẰNG VÀ CÁC HÀM THÀNH VIÊN CONST

Một vài đối tượng cần được thay đổi và một vài đối tượng thì không. Lập trình viên có thể sử dụng từ khóa **const** để cho biết đối tượng không thể thay đổi được, và nếu có cố gắng thay đổi đối tượng thì xảy ra lỗi. Chẳng hạn:

```
const Time Noon(12,0,0); //Khai báo một đối tượng const
```

Các trình biên dịch C++ lưu ý đến các khai báo **const** vì thế các trình biên dịch cấm hoàn toàn bất kỳ hàm thành viên nào gọi các đối tượng **const** (Một vài trình biên dịch chỉ cung cấp một cảnh báo). Điều này thì khắc nghiệt bởi vì các client của đối tượng hầu như chắc chắn sẽ muốn sử dụng các hàm thành viên "get" khác nhau với đối tượng, và tất nhiên không thể thay đổi đối tượng. Để cung cấp cho điều này, lập trình viên có thể khai báo các hàm thành viên **const**; điều này chỉ có thể thao tác trên các đối tượng **const**. Dĩ nhiên các hàm thành viên **const** không thể thay đổi đối tượng - trình biên dịch cấm điều này.

Một hàm được mô tả như **const** khi cả hai trong phần khai báo và trong phần định nghĩa của nó được chèn thêm từ khóa **const** sau danh sách các tham số của hàm, và trong trường hợp của định nghĩa hàm trước dấu ngoặc móc trái ({) mà bắt đầu thân hàm. Chẳng hạn, hàm thành viên của lớp *A* nào đó:

```
int A::GetValue() const
{
return PrivateDataMember;
}
```

Nếu một hàm thành viên **const** được định nghĩa bên ngoài định nghĩa của lớp thì khai báo hàm và định nghĩa hàm phải bao gồm **const** ở mỗi phần.

Một vấn đề nảy sinh ở đây đối với các constructor và destructor, mỗi hàm thường cần thay đổi đối tượng. Khai báo **const** không yêu cầu đối với các constructor và destructor của các đối tượng **const**. Một constructor phải được phép thay đổi một đối tượng mà đối tượng có thể được khởi tạo thích hợp. Một destructor phải có khả năng thực hiện vai trò "công việc kết thúc nội trợ" trước khi đối tượng được hủy.

Ví dụ 3.13: Chương trình sau sử dụng một lớp *Time* với các đối tượng **const** và các hàm thành viên **const**.

```
2: #include <iostream.h>
3: class Time
4: {
5: public:
6: Time(int = 0, int = 0, int = 0); //Constructor mặc định
7: //Các hàm set
8: void SetTime(int, int, int); //Thiết lập thời gian
9: void SetHour(int); //Thiết lập Hour
10: void SetMinute(int); //Thiết lập Minute
11: void SetSecond(int); //Thiết lập Second
12: //Các hàm get
13: int GetHour() const; //Trả về Hour
14: int GetMinute() const; //Trả về Minute
15: int GetSecond() const; //Trả về Second
16: //Các hàm in
17: void PrintMilitary() const; //In t.gian theo dạng giờ quân đội
18: void PrintStandard() const; //In thời gian theo dạng giờ chuẩn
19: private:
20: int Hour; //0 - 23
21: int Minute; //0 - 59
22: int Second; //0 – 59
23: };
24:
25: //Constructor khởi động dữ liệu private
26: //Các giá trị mặc định là 0
27: Time::Time(int hr, int min, int sec)
28: {
29: SetTime(hr, min, sec);
30: }
31:
32: //Thiết lập các giá trị của Hour, Minute, và Second
33: void Time::SetTime(int h, int m, int s)
34: {
35: Hour = (h >= 0 && h < 24) ? h : 0;
36: Minute = (m >= 0 && m < 60) ? m : 0;
37: Second = (s >= 0 && s < 60) ? s : 0;
38: }
39:
40: //Thiết lập giá trị của Hour
41: void Time::SetHour(int h)
42: {
43: Hour = (h >= 0 && h < 24) ? h : 0;
44: }
45:
46: //Thiết lập giá trị của Minute
47: void Time::SetMinute(int m)
48: {
49: Minute = (m >= 0 && m < 60) ? m : 0;
50: }
51:
52: //Thiết lập giá trị của Second
53: void Time::SetSecond(int s)
54: {
55: Second = (s >= 0 && s < 60) ? s : 0;
56: }
57:
```

```
58: //Lấy giá trị của Hour
59: int Time::GetHour() const
60: {
61: return Hour;
62: }
63:
64: //Lấy giá trị của Minute
65: int Time::GetMinute() const
66: {
67: return Minute;
68: }
69:
70: //Lấy giá trị của Second
71: int Time::GetSecond() const
72: {
73: return Second;
74: }
75:
76: //Hiển thị thời gian dạng giờ quân đội: HH:MM:SS
77: void Time::PrintMilitary() const
78: {
79: cout << (Hour < 10 ? "0" : "") << Hour << ":"
80:      << (Minute < 10 ? "0" : "") << Minute << ":"
81:      << (Second < 10 ? "0" : "") << Second;
82: }
83:
84: //Hiển thị thời gian dạng chuẩn: HH:MM:SS AM (hay PM)
85: void Time::PrintStandard() const
86: {
87: cout << ((Hour == 12) ? 12 : Hour % 12) << ":"
88:       << (Minute < 10 ? "0" : "") << Minute << ":"
89:       << (Second < 10 ? "0" : "") << Second
90:       << (Hour < 12 ? " AM" : " PM");
91: }
92:
93: int main()
94: {
95: const Time T(19, 33, 52); //Đối tượng hằng
96: T.SetHour(12); //ERROR: non-const member function
97: T.SetMinute(20); //ERROR: non-const member function
98: T.SetSecond(39); //ERROR: non-const member function
99: return 0;
100: }
```

Chương trình này khai báo một đối tượng hằng của lớp *Time* và cố gắng sửa đổi đối tượng với các hàm thành viên không hằng *SetHour*(), *SetMinute*() và *SetSecond*(). Các lỗi cảnh báo được phát sinh bởi trình biên dịch (Borland C++) như hình 3.13.

```
Warning CT3_13.CPP 99: Non-const function
        Time::SetHour(int) called for const object
Warning CT3_13.CPP 100: Non-const function
        Time::SetMinute(int) called for const object
Warning CT3_13.CPP 101: Non-const function
        Time::SetSecond(int) called for const object
```

Hình 3.13: Các cảnh báo của chương trình ở ví dụ 3.13

Lưu ý: Hàm thành viên **const** có thể được đa năng hóa với một phiên bản non-const. Việc lựa chọn hàm thành viên đa năng hóa nào để sử dụng được tạo một cách tự động bởi trình biên dịch dựa vào nơi mà đối tượng được khai báo **const** hay không.

Một đối tượng **const** không thể được thay đổi bởi phép gán vì thế nó phải được khởi động. Khi một thành viên dữ liệu của một lớp được khai báo **const**, một bộ khởi tạo thành viên (member initializer) phải được sử dụng để cung cấp cho constructor với giá trị ban đầu của thành viên dữ liệu đối với một đối tượng của lớp.

Ví dụ 3.14: C.trình sau sử dụng một bộ khởi tạo thành viên để khởi tạo một hằng của kiểu dữ liệu có sẵn.

```
2: #include <iostream.h>
3: class IncrementClass
4: {
5: public:
6: IncrementClass (int C = 0, int I = 1);
7: void AddIncrement()
8: {
9: Count += Increment;
10: }
11: void Print() const;
12: private:
13: int Count;
14: const int Increment; //Thành viên dữ liệu const
15: };
16:
17: //Constructor của lớp IncrementClass
18: //Bộ khởi tạo với thành viên const
19: IncrementClass::IncrementClass (int C, int I) : Increment(I)
20: {
21: Count = C;
22: }
23:
24: //In dữ liệu
25: void IncrementClass::Print() const
26: {
27: cout << "Count = " << Count
28: #   #   << ", Increment = " << Increment << endl;
30: }
31:
32: int main()
33: {
34: IncrementClass Value(10, 5);
35:
36: cout << "Before incrementing: ";
37: Value.Print();
38: for (int J = 1; J <= 3; J++)
40: {
41: Value.AddIncrement();
42: cout << "After increment " << J << ": ";
43: Value.Print();
44: }
45: return 0;
46: }
```

Chương trình này sử dụng cú pháp bộ khởi tạo thành viên để khởi tạo thành viên dữ liệu **const** *Increment* của lớp *IncrementClass* ở dòng 19.

Chúng ta chạy ví dụ 3.14, kết quả ở hình 3.14

```
Before incrementing: Count = 10, Increment = 5
After increment 1: Count = 15, Increment = 5
After increment 2: Count = 20, Increment = 5
After increment 3: Count = 25, Increment = 5
```

Hình 3.14: Kết quả của ví dụ 3.14

Ký hiệu : Increment(I) (ở dòng 19 của ví dụ 3.14) sinh ra *Increment* được khởi động với giá trị là *I*. Nếu nhiều bộ khởi tạo thành viên được cần, đơn giản bao gồm chúng trong danh sách phân cách dấu phẩy sau dấu hai chấm. Tất cả các thành viên dữ liệu có thể được khởi tạo sử dụng cú pháp bộ khởi tạo thành viên.

Nếu trong ví dụ 3.14 chúng ta cố gắng khởi tạo *Increment* với một lệnh gán hơn là với một bộ khởi tạo thành viên như sau:

```
IncrementClass::IncrementClass (int C, int I)
{   Count = C;
    Increment = I;
}
```

Khi đó trình biên dịch (Borland C++) sẽ có thông báo lỗi như sau:

```
Compiling CT3_14.CPP:
Warning CT3_14.CPP 22: Constant member 'Increment' is not initialized
Error CT3_14.CPP 24: Cannot modify a const object
Warning CT3_14.CPP 25: Parameter 'I' is never used
```

Hình 3.15: Thông báo lỗi khi cố gắng khởi tạo một thành viên dữ liệu **const** bằng phép gán

XIV. LỚP NHƯ LÀ CÁC THÀNH VIÊN CỦA CÁC LỚP KHÁC

Một lớp có thể có các đối tượng của các lớp khác như các thành viên. Khi một đối tượng đi vào phạm vi, constructor của nó được gọi một cách tự động, vì thế chúng ta cần mô tả các tham số được truyền như thế nào tới các constructor của đối tượng thành viên. Các đối tượng thành viên được xây dựng theo thứ tự mà trong đó chúng được khai báo (không theo thứ tự mà chúng được liệt kê trong danh sách bộ khởi tạo thành viên của constructor) và trước các đối tượng của lớp chứa đựng chúng được xây dựng.

Ví dụ 3.15: Chương trình sau minh họa các đối tượng như các thành viên của các đối tượng khác.

```
1: #include <iostream.h>
2: #include <string.h>
3:
4: class Date
5: {
6: public:
7: Date(int = 1, int = 1, int = 1900); //Constructor mặc định
8: void Print() const; //In ngày theo dạng Month/Day/Year
9: private:
10: int Month; //1-12
11: int Day; //1-31
12: int Year; //Năm bất kỳ
13://Hàm tiện ích để kiểm tra Day tương thích đối với Month và Year
14: int CheckDay(int);
15: };
16:
17: class Employee
18: {
```

```
19: public:
20: Employee(char *, char *, int, int, int, int, int, int);
21: void Print() const;
22: private:
23: char LastName[25];
24: char FirstName[25];
25: Date BirthDate;
26: Date HireDate;
27: };
28:
29: //Constructor: xác nhận giá trị tương thích của Month
30: //Gọi hàm CheckDay() để xác nhận giá trị tương thích của Day
31: Date::Date(int Mn, int Dy, int Yr)
32: {
33: if (Mn > 0 && Mn <= 12)
34: Month = Mn;
35: else
36: {
37: Month = 1;
38: cout << "Month " << Mn << " invalid. Set to Month 1."
39:            << endl;
40: }
41: Year = Yr;
42: Day = CheckDay(Dy);
43: cout << "Date object constructor for date ";
44: Print();
45: cout << endl;
46: }
47:
48: //Hàm xác nhận giá trị Day tương thích đưa vào Month và Year
49: int Date::CheckDay(int TestDay)
50: {
51: static int DaysPerMonth[13] = {0, 31, 28, 31, 30, 31,
52: 9; 9; 9; 9; 9; 9; #   #   #   #   30, 31, 31, 30,31, 30, 31};
53:
54: if (TestDay > 0 && TestDay <= DaysPerMonth[Month])
55: return TestDay;
56: if (Month == 2 && TestDay == 29 &&
57: ; ; (Year % 400 == 0 || (Year % 4 == 0 && Year % 100 != 0)))
58: return TestDay;
59: cout << "Day " << TestDay << "invalid. Set to Day 1." << endl;
60: return 1;
61: }
62:
63: //In đối tượng Date dạng Month/Day/Year
64: void Date::Print() const
65: {
66: cout << Month << '/' << Day << '/' << Year;
67: }
68:
69: Employee::Employee(char *FName, char *LName,
70:             int BMonth, int BDay, int BYear,
71:             int HMonth, int HDay, int HYear)
72: :BirthDate(BMonth, BDay, BYear), HireDate(HMonth, HDay, HYear)
73: {
74://Sao chép FName vào FirstName và phải chắc chắn rằng nó phù hợp
75: int Length = strlen(FName);
76:
```

```
77: Length = Length < 25 ? Length : 24;
78: strncpy(FirstName, FName, Length);
79: FirstName[Length] = '\0';
80: //Sao chép LName vào LastName và phải chắc chắn rằng nó phù hợp
81: Length = strlen(LName);
82: Length = Length < 25 ? Length : 24;
83: strncpy(LastName, LName, 24);
84: LastName[Length] = '\0';
85: cout << "Employee object constructor: "
86:       << FirstName << ' ' << LastName << endl;
87: }
88:
89: void Employee::Print() const
90: {
91: cout << LastName << ", " << FirstName << endl << "Hired: ";
92: HireDate.Print();
93: cout << " Birthday: ";
94: BirthDate.Print();
95: cout << endl;
96: }
97:
98: int main()
99: {
100: Employee E("Bob", "Jones", 7, 24, 49, 3, 12, 88);
101:
102 cout << endl;
103: E.Print();
104: cout << endl << "Test Date constructor with invalid values:"
105:       << endl;
106: Date D(14, 35, 94); //Các giá trị Date không hợp lệ
107: return 0;
108: }
```

Chương trình gồm lớp *Employee* chứa các thành viên dữ liệu **private** *LastName*, *FirstName*, *BirthDate* và *HireDate*. Các thành viên *BirthDate* và *HireDate* là các đối tượng của lớp *Date* mà chứa các thành viên dữ liệu private *Month*, *Day* và *Year*. Chương trình khởi tạo một đối tượng *Employee*, và các khởi tạo và các hiển thị các thành viên dữ liệu của nó. Chú ý về cú pháp của phần đầu trong định nghĩa constructor của lớp *Employee*:

```
Employee::Employee(char *FName, char *LName, int BMonth, int BDay, int BYear,
                                                    int HMonth, int HDay, int HYear)
                   :BirthDate(BMonth, BDay, BYear), HireDate(HMonth, HDay, HYear)
```

Constructor lấy tám tham số (*FName*, *LName*, *BMonth*, *BDay*, *BYear*, *HMonth*, *HDay*, và *HYear*). Dấu hai chấm trong phần đầu phân tách các bộ khởi tạo từ danh sách các tham số. Các bộ khởi tạo định rõ các tham số truyền chon constructor của các đối tượng thành viên. Vì thế *BMonth*, *BDay* và *BYear* được truyền cho constructor của đối tượng *BirthDate*, và *HMonth*, *HDay*, và *HYear* được truyền cho constructor của đối tượng *HireDate*. Nhiều bộ khởi tạo được phân tách bởi dấu phẩy.

Chúng ta chạy ví dụ 3.15, kết quả ở hình 3.16

```
Date object constructor for date 7/24/49
Date object constructor for date 3/12/88
Employee object constructor: Bob Jones

Jones, Bob
Hired: 3/12/88  Birthday: 7/24/49

Test Date constructor with invalid values:
Month 14 invalid. Set to Month 1.
Day 35invalid. Set to Day 1.
Date object constructor for date 1/1/94
```

Hình 3.16: Kết quả của ví dụ 3.15

Một đối tượng thành viên không cần được khởi tạo thông qua một bộ khởi tạo thành viên. Nếu một bộ khởi tạo thành viên không được cung cấp, constructor mặc định của đối tượng thành viên sẽ được gọi một cách tự động. Các giá trị nếu có thiết lập bởi constructor mặc định thì có thể được ghi đè bởi các hàm set.

XV. CÁC HÀM VÀ CÁC LỚP friend

Một hàm **friend** của một lớp được định nghĩa bên ngoài phạm vi của lớp đó, lúc này có quyền truy cập đến các thành viên **private** hoặc **protected** của một lớp. Một hàm hay toàn bộ lớp có thể được khai báo là một **friend** của lớp khác.

Để khai báo một hàm là một **friend** của một lớp, đứng trước prototype của hàm trong định nghĩa lớp với từ khóa **friend**. như sau:

friend <*function-declarator*>**;**

Để khai báo một lớp là **friend** của lớp khác như sau:

friend <*class-name*>**;**

Ví dụ 3.16: Chương trình sau minh họa khai báo và sử dụng hàm **friend**.

```
1: #include <iostream.h>
2:
3: class Count
4: {
5: friend void SetX(Count &, int); //Khai báo friend
6: public:
7: Count()//Constructor
8: {
9: X = 0;
10: }
11: void Print() const //Xuất
12: {
13: cout << X << endl;
14: }
15: private:
16: int X;
17: };
18:
19: //Có thể thay đổi dữ liệu private của lớp Count vì
20: //SetX() khai báo là một hàm friend của lớp Count
21: void SetX(Count &C, int Val)
22: {
23: C.X = Val; //Hợp lệ: SetX() là một hàm friend của lớp Count
24: }
```

```
25:
26: int main()
27: {
28: Count Object;
29:
30: cout << "Object.X after instantiation: ";
31: Object.Print();
32: cout << "Object.X after call to SetX friend function: ";
33: SetX(Object, 8); //Thiết lập X với một friend
34: Object.Print();
35: return 0;
36: }
```

Chúng ta chạy ví dụ 3.16, kết quả ở hình 3.17

```
Object.X after instantiation: 0
Object.X after call to SetX friend function: 8
```

Hình 3.17: Kết quả của ví dụ 3.16

Có thể chỉ định các hàm được đa năng hóa là các **friend** của lớp. Mỗi hàm được đa năng hóa phải được khai báo tường minh trong định nghĩa lớp như là một **friend** của lớp.

XVI. CON TRỎ THIS

Khi một hàm thành viên tham chiếu thành viên khác của lớp cho đối tượng cụ thể của lớp đó, làm thế nào C++ bảo đảm rằng đối tượng thích hợp được tham chiếu? Câu trả lời là mỗi đối tượng duy trì một con trỏ trỏ tới chính nó – gọi là con trỏ **this** – Đó là một tham số ẩn trong tất cả các tham chiếu tới các thành viên bên trong đối tượng đó. Con trỏ **this** cũng có thể được sử dụng tường minh. Mỗi đối tượng có thể xác định địa chỉ của chính mình bằng cách sử dụng từ khóa **this**.

Con trỏ **this** được sử dụng để tham chiếu cả các thành viên dữ liệu và hàm thành viên của một đối tượng. Kiểu của con trỏ **this** phụ thuộc vào kiểu của đối tượng và trong hàm thành viên con trỏ **this** được sử dụng là khai báo **const**. Chẳng hạn, một hàm thành viên không hằng của lớp *Employee* con trỏ **this** có kiểu là:

Employee * const //Con trỏ hằng trỏ tới đối tượng Employee

Đối với một hàm thành viên hằng của lớp *Employee* con trỏ **this** có kiểu là:

const Employee * const //Con trỏ hằng trỏ tới đối tượng Employee mà là một hằng

Ví dụ 3.17: Chương trình sau minh họa sử dụng tường minh của con trỏ **this** để cho phép một hàm thành viên của lớp *Test* in dữ liệu *X* của một đối tượng *Test*.

```
1: #include <iostream.h>
2:
3: class Test
4: {
5: public:
6: Test(int = 0); // Constructor mặc định
7: void Print() const;
8: private:
9: int X;
10: };
11:
12: Test::Test(int A)
13: {
14: X = A;
15: }
16:
```

```
17: void Test::Print() const
18: {
19: cout << " X = " << X << endl
20:       << " this->X = " << this->X << endl
21:       << "(*this).X = " << (*this).X << endl;
22: }
23:
24: int main()
25: {
26: Test A(12);
27:
28: A.Print();
29: return 0;
30: }
```

Chúng ta chạy ví dụ 3.17, kết quả ở hình 3.18

```
    X = 12
 this->X = 12
(*this).X = 12
```

Hình 3.18: Kết quả của ví dụ 3.17

Một cách khác sử dụng con trỏ **this** là cho phép móc vào nhau các lời gọi hàm thành viên.

Ví dụ 3.18: Chương trình sau minh họa trả về một tham chiếu tới một đối tượng *Time* để cho phép các lời gọi hàm thành viên của lớp *Time* được móc nối vào nhau.

```
1: #include <iostream.h>
2:
3: class Time
4: {
5: public:
6: Time(int = 0, int = 0, int = 0); // Constructor mặc định
7: // Các hàm set
8: Time &SetTime(int, int, int); // Thiết lập Hour, Minute va
Second
9: Time &SetHour(int); // Thiết lập Hour
10: Time &SetMinute(int); // Thiết lập Minute
11: Time &SetSecond(int); // Thiết lập Second
12: // Các hàm get
13: int GetHour() const; // Trả về Hour
14: int GetMinute() const; // Trả về Minute
15: int GetSecond() const; // Trả về Second
16: // Các hàm in
17: void PrintMilitary() const; // In t.gian theo dạng giờ quân đội
18: void PrintStandard() const; // In thời gian theo dạng giờ chuẩn
19: private:
20: int Hour; // 0 - 23
21: int Minute; // 0 - 59
22: int Second; // 0 - 59
23: };
24:
25: // Constructor khởi động dữ liệu private
26: // Gọi hàm thành viên SetTime() để thiết lập các biến
27: // Các giá trị mặc định là 0
28: Time::Time(int Hr, int Min, int Sec)
29: {
30: SetTime(Hr, Min, Sec);
```

```
31: }
32:
33: // Thiết lập các giá trị của Hour, Minute, và Second
34: Time &Time::SetTime(int H, int M, int S)
35: {
36: Hour = (H >= 0 && H < 24) ? H : 0;
37: Minute = (M >= 0 && M < 60) ? M : 0;
38: Second = (S >= 0 && S < 60) ? S : 0;
39: return *this; // Cho phép móc nối
40: }
41:
42: // Thiết lập giá trị của Hour
43: Time &Time::SetHour(int H)
44: {
45: Hour = (H >= 0 && H < 24) ? H : 0;
46: return *this; // Cho phép móc nối
47: }
48:
49: // Thiết lập giá trị của Minute
50: Time &Time::SetMinute(int M)
51: {
52: Minute = (M >= 0 && M < 60) ? M : 0;
53: return *this; // Cho phép móc nối
54: }
55:
56: // Thiết lập giá trị của Second
57: Time &Time::SetSecond(int S)
58: {
59: Second = (S >= 0 && S < 60) ? S : 0;
60: return *this; // Cho phép móc nối
61: }
62:
63: // Lấy giá trị của Hour
64: int Time::GetHour() const
65: {
66: return Hour;
67: }
68:
69: // Lấy giá trị của Minute
70: int Time::GetMinute() const
71: {
72: return Minute;
73: }
74:
75: // Lấy giá trị của Second
76: int Time::GetSecond() const
77: {
78: return Second;
79: }
80:
81: // Hiển thị thời gian dạng giờ quân đội: HH:MM:SS
82: void Time::PrintMilitary() const
83: {
84: cout << (Hour < 10 ? "0" : "") << Hour << ":"
85:       << (Minute < 10 ? "0" : "") << Minute << ":"
86:       << (Second < 10 ? "0" : "") << Second;
87: }
88:
```

```
89: // Hiển thị thời gian dạng chuẩn: HH:MM:SS AM (hay PM)
90: void Time::PrintStandard() const
91: {
92: cout << ((Hour == 0 || Hour == 12) ? 12 : Hour % 12) << ":"
93:       << (Minute < 10 ? "0" : "") << Minute << ":"
94:       << (Second < 10 ? "0" : "") << Second
95:       << (Hour < 12 ? " AM" : " PM");
96: }
97:
100: int main()
101: {
102: Time T;
103:
104: // Các lời gọi móc nối vào nhau
105: T.SetHour(18).SetMinute(30).SetSecond(22);
106: cout << "Military time: ";
107: T.PrintMilitary();
108: cout << endl << "Standard time: ";
109: T.PrintStandard();
110: cout << endl << endl << "New standard time: ";
111: // Các lời gọi móc nối vào nhau
112: T.SetTime(20, 20, 20).PrintStandard();
113: cout << endl;
114: return 0;
115: }
```

Các hàm thành viên *SetTime*(), *SetHour*(), *SetMinute*() và *SetSecond*() mỗi hàm đều trả về ***this** với kiểu trả về là *Time* &. Toán tử chấm liên kết từ trái sang phải, vì vậy biểu thức:

T.SetHour(18).SetMinute(30).SetSecond(22);

Đầu tiên gọi *T.SetHour(18)* thì trả về một tham chiếu tới đối tượng *T* là giá trị của lời gọi hàm này. Phần còn lại của biểu thức được hiểu như sau:

T.SetMinute(30).SetSecond(22);

T.SetMinute(30) gọi thực hiện và trả về tương đương của *T*. Phần còn của biểu thức là:

T.SetSecond(22);

Chúng ta chạy ví dụ 3.18, kết quả ở hình 3.19

```
Military time: 18:30:22
Standard time: 6:30:22 PM

New standard time: 8:20:20 PM
```

Hình 3.19: Kết quả của ví dụ 3.18

XVII. CÁC ĐỐI TƯỢNG ĐƯỢC CẤP PHÁT ĐỘNG

Các đối tượng có thể được cấp phát động giống như các dữ liệu khác bằng toán tử **new** và **delete**. Chẳng hạn:

Time *TimePtr = new Time(1,20,26);

……..

delete TimePtr;

Toán tử **new** tự động gọi hàm constructor ,và toán tử **delete** tự động gọi destructor.

XVIII. CÁC THÀNH VIÊN TĨNH CỦA LỚP

Bình thường, mỗi đối tượng của một lớp có bản sao chép của chính nó của tất cả các thành viên dữ liệu của lớp. Trong các trường hợp nhất định chỉ có duy nhất một bản chép thành viên dữ liệu đặc biệt cần phải dùng chung bởi tất cả các đối tượng của một lớp. Một thành viên dữ liệu tĩnh được sử dụng cho những điều đó và các lý do khác. Một thành viên dữ liệu tĩnh biểu diễn thông tin toàn lớp (class-wide). Khai báo một thành viên tĩnh bắt đầu với từ khóa **static**.

Mặc dù các thành viên dữ liệu tĩnh có thể giống như các biến toàn cục, các thành viên dữ liệu tĩnh có phạm vi lớp. Các thành viên tĩnh có thể là **public**, **private** hoặc **protected**. Các thành viên dữ liệu tĩnh phải được khởi tạo một lần (và chỉ một lần) tại phạm vi file. Các thành viên lớp tĩnh **public** có thể được truy cập thông qua bất kỳ đối tượng nào của lớp đó, hoặc chúng có thể được truy cập thông qua tên lớp sử dụng toán tử định phạm vi. Các thành viên lớp tĩnh **private** và **protected** phải được truy cập thông qua các hàm thành viên **public** của lớp hoặc thông qua các **friend** của lớp. Các thành viên lớp tĩnh tồn tại ngay cả khi đối tượng của lớp đó không tồn tại. Để truy cập một thành viên lớp tĩnh **public** khi các đối tượng của lớp không tồn tại, đơn giản thêm vào đầu tên lớp và toán tử định phạm vi cho thành viên dữ liệu. Để truy cập một thành viên lớp tĩnh **private** hoặc **protected** khi các đối tượng của lớp không tồn tại, một hàm thành viên **public** phải được cung cấp và hàm phải được gọi bởi thêm vào đầu tên của nó với tên lớp và toán tử định phạm vi.

Ví dụ 3.19: Chương trình sau minh họa việc sử dụng thành viên dữ liệu tĩnh **private** và một hàm thành viên tĩnh **public**.

```
1: #include <iostream.h>
2: #include <string.h>
3: #include <assert.h>
4:
5: class Employee
6: {
7: public:
8: Employee(const char*, const char*); // Constructor
9: ~Employee(); // Destructor
10: char *GetFirstName() const; // Trả về first name
11: char *GetLastName() const; // Trả về last name
12: // Hàm thành viên tĩnh
13: static int GetCount(); // Trả về số đối tượng khởi tạo
14: private:
15: char *FirstName;
16: char *LastName;
17: // static data member
18: static int Count; // Số đối tượng khởi tạo
19: };
20:
21: // Khởi tạo thành viên dữ liệu tĩnh
22: int Employee::Count = 0;
23:
24://Định nghĩa hàm thành viên tỉnh mà trả về số đối tượng khởi tạo
25: int Employee::GetCount()
26: {
27: return Count;
28: }
29:
30: // Constructor cấp phát động cho first name và last name
31: Employee::Employee(const char *First, const char *Last)
32: {
33: FirstName = new char[ strlen(First) + 1 ];
34: assert(FirstName != 0); // Bảo đảm vùng nhớ được cấp phát
35: strcpy(FirstName, First);
36: LastName = new char[ strlen(Last) + 1 ];
```

```
37: assert(LastName != 0); // Bảo đảm vùng nhớ được cấp phát
38: strcpy(LastName, Last);
39: ++Count; // Tăng số đối tượng lên 1
40: cout << "Employee constructor for " << FirstName
41:       << ' ' << LastName << " called." << endl;
42: }
43:
44: // Destructor giải phóng vùng nhớ đã cấp phát
45: Employee::~Employee()
46: {
47: cout << "~Employee() called for " << FirstName
48:       << ' ' << LastName << endl;
49: delete FirstName;
50: delete LastName;
51: --Count; // Giảm số đối tượng xuống 1
52: }
53:
54: // Trả về first name
55: char *Employee::GetFirstName() const
56: {
57: char *TempPtr = new char[strlen(FirstName) + 1];
58: assert(TempPtr != 0); // Bảo đảm vùng nhớ được cấp phát
59: strcpy(TempPtr, FirstName);
60: return TempPtr;
61: }
62:
63: // Trả về last name
64: char *Employee::GetLastName() const
65: {
66: char *TempPtr = new char[strlen(LastName) + 1];
67: assert(TempPtr != 0); // Bảo đảm vùng nhớ được cấp phát
68: strcpy(TempPtr, LastName);
69: return TempPtr;
70: }
71:
72: int main()
73: {
74: cout << "Number of employees before instantiation is "
75:       << Employee::GetCount() << endl; // Sử dụng tên lớp
76: Employee *E1Ptr = new Employee("Susan", "Baker");
77: Employee *E2Ptr = new Employee("Robert", "Jones");
78: cout << "Number of employees after instantiation is "
79:       << E1Ptr->GetCount() << endl;
80: cout << endl << "Employee 1: "
81:      << E1Ptr->GetFirstName()
82:      << " " << E1Ptr->GetLastName()
83:      << endl << "Employee 2: "
84:      << E2Ptr->GetFirstName()
85:      << " " << E2Ptr->GetLastName() << endl << endl;
86: delete E1Ptr;
87: delete E2Ptr;
88: cout << "Number of employees after deletion is "
89:      << Employee::GetCount() << endl;
90: return 0;
91: }
```

Thành viên dữ liệu *Count* được khởi tạo là zero ở phạm vi file với lệnh:

```
int Employee::Count = 0;
```

Thành viên dữ liệu Count duy trì số các đối tượng của lớp *Employee* đã được khởi tạo. Khi đối tượng của lớp *Employee* tồn tại, thành viên *Count* có thể được tham chiếu thông qua bất kỳ hàm thành viên nào của một đối tượng *Employee* – trong ví dụ này, *Count* được tham chiếu bởi cả constructor lẫn destructor. Khi các đối tượng của lớp *Employee* không tồn tại, thành viên *Count* có thể vẫn được tham chiếu nhưng chỉ thông qua một lời gọi hàm thành viên tĩnh public *GetCount*() như sau:

```
Employee::GetCount()
```

Hàm *GetCount*() được sử dụng để xác định số các đối tượng của *Employee* khởi tạo hiện hành. Chú ý rằng khi không có các đối tượng trong chương trình, lời gọi hàm *Employee::GetCount*() được đưa ra. Tuy nhiên khi có các đối tượng khởi động hàm *GetCount*() có thể được gọi thông qua một trong các đối tượng như sau:

```
E1Ptr->GetCount()
```

Trong chương trình ở các dòng 34, 37, 58 và 67 sử dụng hàm **assert()** (định nghĩa trong assert.h). Hàm này kiểm tra giá trị của biểu thức. Nếu giá trị của biểu thức là 0 (false), hàm **assert()** in một thông báo lỗi và gọi hàm **abort()** (định nghĩa trong stdlib.h) để kết thúc chương trình thực thi. Nếu biểu thức có giá trị khác 0 (true) thì chương trình tiếp tục. Điều này rất có ích cho công cụ debug đối với việc kiểm tra nếu một biến có giá trị đúng. Chẳng hạn hàm ở dòng 34 hàm **assert()** kiểm tra con trỏ *FirstName* để xác định nếu nó không bằng 0 (null). Nếu điều kiện trong khẳng định (assertion) cho trước là đúng, chương trình tiếp tục mà không ngắt. Nếu điều kiện trong khẳng định cho trước là sai, một thông báo lỗi chứa số dòng, điều kiện được kiểm tra, và tên file trong đó sự khẳng định xuất hiện được in, và chương trình kết thúc. Khi đó lập trình viên có thể tập trung vào vùng này của đoạn mã để tìm lỗi.

Các khẳng định không phải xóa từ chương trình khi debug xong. Khi các khẳng định không còn cần thiết cho mục đích debug trong một chương trình, dòng sau:

```
#define NDEBUG
```

được thêm vào ở đầu file chương trình. Điều này phát sinh tiền xử lý bỏ qua tất cả các khẳng định thay thế cho lập trình viên xóa mỗi khẳng định bằng tay.

Chúng ta chạy ví dụ 3.19, kết quả ở hình 3.20

```
Number of employees before instantiation is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after instantiation is 2

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Susan Baker
~Employee() called for Robert Jones
Number of employees after deletion is 0
```

Hình 3.20: Kết quả của ví dụ 3.19

Một hàm thành viên có thể được khai báo là **static** nếu nó không truy cập đến các thành viên không tĩnh. Không giống như các thành viên không tĩnh, một hàm thành viên tĩnh không có con trỏ **this** bởi vì các thành viên dữ liệu tĩnh và các hàm thành viên tĩnh tồn tại độc lập với bất kỳ đối tượng nào của lớp.

Chú ý: Hàm thành viên dữ liệu tĩnh không được là **const**.

BÀI TẬP

Bài 1: Xây dựng lớp Stack, dữ liệu bao gồm đỉnh stack và vùng nhớ của stack. Các thao tác gồm:

- Khởi động stack.
- Kiểm tra stack có rỗng không?
- Kiểm tra stack có đầy không?
- Push và pop.

Bài 2: Xây dựng lớp hình trụ Cylinder, dữ liệu bao gồm bán kính và chiều cao của hình trụ. Các thao tác gồm hàm tính diện tích toàn phần và thể tích của hình trụ đó.

Bài 3: Hãy xây dựng một lớp Point cho các điểm trong không gian ba chiều (x,y,z). Lớp chứa một constructor mặc định, một hàm Negate() để biến đổi điểm thành đại lượng có dấu âm, một hàm Norm() trả về khoảng cách từ gốc và một hàm Print().

Bài 4: Xây dựng một lớp Matrix cho các ma trận bao gồm một constructor mặc định, hàm xuất ma trận, nhập ma trận từ bàn phím, cộng hai ma trận, trừ hai ma trận và nhân hai ma trận.

Bài 5: Xây dựng một lớp Matrix cho các ma trận vuông bao gồm một constructor mặc định, hàm xuất ma trận, tính định thức và tính ma trận nghịch đảo.

Bài 6: Xây dựng lớp Person để quản lý họ tên, năm sinh, điểm chín môn học của tất cả các học viên của lớp học. Cho biết bao nhiêu học viên trong lớp được phép làm luận văn tốt nghiệp, bao nhiêu học viên thi tốt nghiệp, bao nhiêu người phải thi lại và tên môn thi lại. Tiêu chuẩn để xét:

- Làm luận văn phải có điểm trung bình lớn hơn 7 trong đó không có môn nào dưới 5.
- Thi tốt nghiệp khi điểm trung bình không lớn hơn 7 và điểm các môn không dưới 5.
- Thi lại có môn dưới 5.

Bài 7: Xây dựng một lớp String. Mỗi đối tượng của lớp String sẽ đại diện một chuỗi ký tự. Các thành viên dữ liệu là chiều dài chuỗi và chuỗi ký tự thực. Ngoài constructor và destructor còn có các phương thức như tạo một chuỗi với chiều dài cho trước, tạo một chuỗi từ một chuỗi đã có.

Bài 8: Xây dựng một lớp Vector để lưu trữ vector gồm các số thực. Các thành viên dữ liệu gồm:

- Kích thước vector.
- Một mảng động chứa các thành phần của vector.

Ngoài constructor và destructor, còn có các phương thức tính tích vô hướng của hai vector, tính chuẩn của vector (theo chuẩn bất kỳ nào đó).

Bài 9: Xây dựng lớp Employee gồm họ tên và chứng minh nhân dân. Ngoài constructor còn có phương thức nhập, xuất họ tên và chứng minh nhân dân ra màn hình.

CHƯƠNG 4

ĐA NĂNG HÓA TOÁN TỬ

I. DẪN NHẬP

Trong chương 3, chúng ta đã tìm hiểu các điều cơ bản của các lớp C++ và khái niệm kiểu dữ liệu trừu tượng (ADTs). Các thao tác trên các đối tượng của lớp (nghĩa là các thực thể của ADTs) được thực hiện bởi gởi các thông điệp (dưới dạng các lời gọi hàm thành viên) tới các đối tượng. Ký pháp gọi hàm này thì cồng kềnh cho các loại lớp nhất định, đặc biệt là các lớp toán học. Đối với các loại lớp này sẽ là đẹp để sử dụng tập các toán tử có sẵn phong phú của C++ để chỉ rõ các thao tác của đối tượng. Trong chương này tìm hiểu làm thế nào cho phép các toán tử của C++ làm việc với các đối tượng của lớp. Xử lý này được gọi là đa năng hóa toán tử (operator overloading).

Toán tử << được sử dụng nhiều mục đích trong C++ đó là toán tử chèn dòng (stream-insertion) và toán tử dịch chuyển trái. Đây là một ví dụ của đa năng hóa toán tử. Tương tự >> cũng được đa năng hóa. Nó được sử dụng vừa toán tử trích dòng (stream-extraction) và toán tử dịch chuyển phải.

C++ cho phép các lập trình viên đa năng hóa hầu hết các toán tử để biểu thị ngữ cảnh mà trong đó chúng được sử dụng. Trình biên dịch phát sinh đoạn mã thích hợp dựa trên kiểu mà trong đó toán tử được sử dụng. Một vài toán tử được đa năng hóa thường xuyên, đặc biệt là toán tử gán và các toán tử số học như + và -. Công việc thực hiện bởi đa năng hóa các toán tử cũng có thể được thực hiện bởi các lời gọi hàm tường minh, nhưng ký pháp thường sử dụng dễ dàng để đọc.

II. CÁC NGUYÊN TẮC CƠ BẢN CỦA ĐA NĂNG HÓA TOÁN TỬ

Lập trình viên có thể sử dụng các kiểu có sẵn và có thể định nghĩa các kiểu mới. Các kiểu có thể được sử dụng với tập các toán tử phong phú. Các toán tử cung cấp cho các lập trình viên với ký pháp ngắn ngọn cho việc biểu thị các thao tác của đối tượng của các kiểu có sẵn.

Các lập trình viên có thể sử dụng các toán tử với các kiểu do người dùng định nghĩa. Mặc dù C++ không cho phép các toán tử mới được tạo, nó cho phép các toán tử đã tồn tại được đa năng hóa sao cho khi các toán tử này được sử dụng với các đối tượng của lớp, các toán tử có ý nghĩa thích hợp các kiểu mới. Đây chính là một đặc điểm mạnh của C++.

Các toán tử được đa năng hóa bằng cách viết một định nghĩa hàm (bao gồm phần đầu và thân) như khi chúng ta viết một hàm bình thường, ngoại trừ tên hàm bây giờ trở thành từ khóa **operator** theo sau bởi ký hiệu của toán tử được đa năng hóa. Prototype của nó có dạng như sau:

type **operator** operator_symbol (parameter_list);

Để sử dụng một toán tử một các đối tượng của lớp, toán tử phải được đa năng hóa ngoại trừ hai điều. Điều thứ nhất toán tử gán có thể sử dụng với mọi lớp mà không cần đa năng hóa. Cách cư xử mặc định của toán tử gán là một phép gán thành viên của các thành viên dữ liệu của lớp. Chúng ta nhận thấy rằng sao chép thành viên mặc định thì nguy hiểm đối với các lớp với các thành viên mà được cấp phát động. Chúng ta sẽ đa năng hóa một cách tường minh toán tử gán đối với các lớp như thế. Điều thứ hai toán tử địa chỉ (&) cũng có thể được sử dụng với các đối tượng của bất kỳ lớp nào mà không cần đa năng hóa; Nó trả về địa chỉ của đối tượng trong bộ nhớ. Toán tử địa chỉ cũng có thể được đa năng hóa.

III. CÁC GIỚI HẠN CỦA ĐA NĂNG HÓA TOÁN TỬ

Phần lớn các toán tử của C++ có thể được đa năng hóa. Hình 4.1 cho thấy các toán tử có thể được đa năng hóa và hình 4.1 là các toán tử không thể đa năng hóa.

+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete

Hình 4.1:

Các toán tử có thể được đa năng hóa

.	.*	::	?:	sizeof

Hình 4.2: Các toán tử không thể đa năng hóa

Chú ý rằng toán tử ngoặc tròn **()** trong bảng 4.1 là toán tử gọi hàm. Vì toán tử này đứng sau tên hàm có thể chứa trong nó nhiều tham số do đó toán tử ngoặc tròn là một toán tử nhiều ngôi.

Thứ tự ưu tiên của một toán tử không thể được thay đổi bởi đa năng hóa. Điều này có thể dẫn tới các tình trạng bất tiện trong đó một toán tử được đa năng hóa theo một cách đối với mức độ ưu tiên cố định của nó thì không thích hợp. Tuy nhiên, các dấu ngoặc đơn có thể được sử dụng để đặt thứ tự ước lượng của các toán tử đã đa năng hóa trong một biểu thức.

Tính kết hợp của một toán tử không thể được thay đổi bởi đa năng hóa. Các tham số mặc định không thể sử dụng với một toán tử đa năng hóa.

Không thể thay đổi số các toán hạng mà một toán tử yêu cầu: Đa năng hóa các toán tử một ngôi vẫn là các toán tử một ngôi; đa năng hóa các toán tử hai ngôi vẫn là các toán tử hai ngôi. Toán tử ba ngôi duy nhất (**?:**) của C++ không thể đa năng hóa. Các toán tử **&**, *****, **+** và **–** mỗi toán tử có các phiên bản một và hai ngôi.; Các phiên bản một và hai ngôi này có thể được đa năng hóa riêng biệt.

Ý nghĩa của làm sao một toán tử làm việc trên các đối tượng của các kiểu có sẵn không thể thay đổi bởi việc đa năng hóa toán tử. Chẳng hạn, lập trình viên không thể thay đổi ý nghĩa của làm sao toán tử (+) cộng hai số nguyên. Việc đa năng hóa toán tử chỉ làm việc với các đối tượng của các kiểu do người dùng định nghĩa hoặc với một sự pha trộn của một đối tượng của kiểu do người dùng định nghĩa và một đối tượng của một kiểu có sẵn.

Đa năng hóa một toán tử gán và một toán tử cộng để cho phép các lệnh như là:

```
object2 = object2 + object1
```

không bao hàm toán tử += cũng được đa năng hóa để phép các lệnh như là:

```
object2 += object1
```

Hành vi như thế có thể được thực hiện bởi việc đa năng hóa rõ ràng toán tử += cho lớp đó.

IV. CÁC HÀM TOÁN TỬ CÓ THỂ LÀ CÁC THÀNH VIÊN CỦA LỚP HOẶC KHÔNG LÀ CÁC THÀNH VIÊN

Các hàm toán tử có thể là các hàm thành viên hoặc hàm không thành viên; hàm không thành viên thường là các hàm **friend**. Các hàm thành viên sử dụng ngầm con trỏ **this** để chứa một trong các tham số đối tượng lớp của chúng. Tham số lớp đó phải được liệt kê một cách tường minh trong lời gọi hàm không thành viên.

Khi đa năng hóa **()**, **[]**, **->** hoặc =, hàm đa năng hóa toán tử phải được khai báo như một thành viên lớp. Đối với các toán tử khác, các hàm đa năng hóa toán tử có thể là các hàm không thành viên (thường là các hàm **friend**).

Liệu có phải một hàm toán tử được cài đặt như một hàm thành viên hoặc như hàm không thành viên, toán tử vẫn còn được sử dụng cùng cách trong biểu thức. Như vậy cách là cách cài đặt nào tốt nhất?

Khi một hàm toán tử được cài đặt như một hàm thành viên, toán hạng cực trái phải là một đối tượng lớp của toán tử. Nếu toán hạng bên trái phải là một đối tượng của lớp khác hoặc một kiểu có sẵn thì hàm toán tử này phải được cài đặt như hàm không thành viên. Một hàm toán tử cài đặt như hàm không thành viêân cần là một **friend** nếu hàm phải truy cập đến các thành viên **private** hoặc **protected**.

Các hàm thành viên chỉ được gọi khi toán hạng trái của một toán tử hai ngôi là một đối tượng cụ thể của lớp đó, hoặc khi toán hạng đơn của một toán tử một ngôi là một đối tượng của lớp đó.

Ví dụ 4.1: Chúng ta xây dựng lớp số phức với tên lớp là *Complex* và đa năng hóa toán tử + trên lớp này.

```
1: #include <iostream.h>
2:
3: class Complex
4: {
5: private:
6: double Real, Imaginary;
7: public:
8: Complex(double R=0.0,double I=0.0);// Constructor mặc định
9: void Print(); // Hiển thị số phức
10: Complex operator+(Complex Z); // Phép cộng giữa hai số phức
11: Complex operator+(double R); //cộng một số phức với một số thực
12: };
13:
14: Complex::Complex(double R,double I)
15: {
16: Real = R;
17: Imaginary = I;
18: }
19:
20: void Complex::Print()
21: {
22: cout<<'('<<Real<<','<<Imaginary<<')';
23: }
24:
25: Complex Complex::operator + (Complex Z)
26: {
27: Complex Tmp;
28: Tmp.Real = Real + Z.Real;
29: Tmp.Imaginary = Imaginary + Z.Imaginary;
30: return Tmp;
31: }
32:
33: Complex Complex::operator + (double R)
34: {
35: Complex Tmp;
36: Tmp.Real = Real + R;
37: Tmp.Imaginary = Imaginary;
38: return Tmp;
39: }
40:
41: int main()
42: {
43: Complex X,Y(4.3,8.2),Z(3.3,1.1);
44: cout<<"X: ";
45: X.Print();
46: cout<<endl<<"Y: ";
47: Y.Print();
48: cout<<endl<<"Z: ";
49: Z.Print();
50: X = Y + Z;
51: cout<<endl<<endl<<"X = Y + Z:"<<endl;
52: X.Print();
53: cout<<" = ";
54: Y.Print();
55: cout<<" + ";
56: Z.Print();
```

```
57: X = Y + 3.5;
58: cout<<endl<<endl<<"X = Y + 3.5:"<<endl;
59: X.Print();
60: cout<<" = ";
61: Y.Print();
62: cout<<" + 3.5";
63: return 0;
64: }
```

Hàm thành viên toán tử **operator + ()** (từ dòng 25 đến 31 và từ dòng 33 đến 39) trả về một đối tượng có kiểu *Complex* là tổng của hai số phức hoặc tổng của một số phức với một số thực. Chú ý rằng đối tượng tam thời *Tmp* được dùng bên trong hàm **operator + ()** để giữ kết quả, và đó là đối tượng được trả về.

Chúng ta chạy ví dụ 4.1, kết quả ở hình 4.3

```
X: (0,0)
Y: (4.3,8.2)
Z: (3.3,1.1)

X = Y + Z:
(7.6,9.3) = (4.3,8.2) + (3.3,1.1)

X = Y + 3.5:
(7.8,8.2) = (4.3,8.2) + 3.5
```

Hình 4.3: Kết quả của ví dụ 4.1

Do đa năng hóa toán tử + trên lớp *Complex* ở ví dụ 4.1, chúng ta có thể viết:

```
X = Y + Z;
```

Câu lệnh này được trình biên dịch hiểu:

```
X = Y.operator + (Z);
```

Như vậy, trong biểu thức *Y + Z* đối tượng bên trái toán tử + (là đối tượng *Y*) là đối tượng mà qua đó, hàm thành viên toán tử **operator + ()** được gọi. Do đó hàm thành viên toán tử + chỉ nhận một tham số là đối tượng bên phải toán tử và đối tượng bên trái toán tử là đối tượng tạo lời gọi cho hàm toán tử và được truyền bởi con trỏ **this**.

Hàm **operator + ()** trả về một đối tượng *Complex*. Do vậy chúng ta có thể viết:

```
(Y + Z).Print();
```

để in trên màn hình số phức của đối tượng được trả về. Đối tượng do *Y + Z* sinh ra như vậy là một đối tượng tạm thời. Nó sẽ không tồn tại khi hàm thành *Print()* kết thúc.

Hơn nữa khi trả về một đối tượng, toán tử + cho phép một chuỗi phép cộng. Nên chúng ta cũng có thể viết:

```
X = X + Y + Z;
```

Tuy nhiên chúng ta không thể nào viết được câu lệnh sau:

```
X = 3.5 + Y; // Lỗi !!!
```

Chính vì lý do này chúng ta chọn một hàm không thành viên để đa năng hóa một toán tử để cho phép toán tử được giao hoán. Chú ý rằng hàm không thành viên không cần thiết phải là hàm **friend** nếu các hàm set và get thích hợp tồn tại trong phần giao diện public, và đặt biệt nhất nếu các hàm set và get là các hàm **inline**.

Để đa năng hóa toán tử << phải có một toán hạng trái của kiểu **ostream &** (như là **cout** trong biểu thức cout<<X), vì thế nó phải là hàm không thành viên. Tương tự, đa năng hóa toán tử >> phải có một toán hạng trái của kiểu **istream &** (như là **cin** trong biểu thức cin>>X), vì thế vì thế nó cũng phải là hàm không thành viên.

Ngoại trừ đa năng hóa toán tử **>>** và **<<** liên quan đến dòng nhập/xuất dữ liệu chúng ta có hình 4.4 về cách đa năng hóa toán tử như sau:

Biểu thức	Hàm thành viên	Hàm không thành viên
a#b	a.operator#(b)	operator#(a,b)
#a	a.operator()	operator#(a)
a=b	a.operator=(b)	
a[b]	a.operator[](b)	
a(b)	a.operator()(b)	
a->	a.operator->()	
a++	a.operator++(0)	operator++(a,0)
a--	a.operator--(0)	operator--(a,0)

Hình 4.4: Việc cài đặt các hàm toán tử

V. ĐA NĂNG HOÁ CÁC TOÁN TỬ HAI NGÔI

Các toán tử hai ngôi được đa năng hóa trong hình 4.5 sau:

Toán tử	Ví dụ	Toán tử	Ví dụ	Toán tử	Ví dụ
+	a+b	+=	a+=b	**<<=**	a<<=b
-	a-b	**-=**	a-=b	**==**	a==b
*	a*b	*=	a*=b	**!=**	a!=b
/	a/b	/=	a/=b	**<=**	a<=b
%	a%b	**%=**	a%=b	**>=**	a>=b
^	a^b	^=	a^=b	**&&**	a&&b
&	a&b	**&=**	a&=b	||	a||b
|	a|b	|=	a|=b	**,**	a,b
=	a=b	<<	a<<b	**[]**	a[b]
<	a<b	>>	a>>b	**->***	a->*b
>	a>b	>>=	a>>=b		

Hình 4.5: Các toán tử hai ngôi được đa năng hóa

Một toán tử hai ngôi có thể được đa năng hóa như là hàm thành viên không tĩnh với một tham số hoặc như một hàm không thành viên với hai tham số (một trong các tham số này phải là hoặc là một đối tượng lớp hoặc là một tham chiếu đến đối tượng lớp).

Ví dụ 4.2: Chúng ta xây dựng lớp số phức với tên lớp là *Complex* và đa năng hóa các toán tử tính toán + - += -= và các toán tử so sánh == != > >= < <= với các hàm toán tử là các hàm thành viên.

```
1: #include <iostream.h>
2: #include <math.h>
3:
4: class Complex
5: {
6: private:
7: double Real, Imaginary;
8: public:
9: Complex(); // Constructor mặc định
10: Complex(double R,double I);
```

```
11: Complex (const Complex & Z); // Constructor sao chép
12: Complex (double R); // Constructor chuyển đổi
13: void Print(); // Hiển thị số phức
14: // Các toán tử tính toán
15: Complex operator + (Complex Z);
16: Complex operator - (Complex Z);
17: Complex operator += (Complex Z);
18: Complex operator -= (Complex Z);
19: // Các toán tử so sánh
20: int operator == (Complex Z);
21: int operator != (Complex Z);
22: int operator > (Complex Z);
23: int operator >= (Complex Z);
24: int operator < (Complex Z);
25: int operator <= (Complex Z);
26: private:
27: double Abs(); // Giá trị tuyệt đối của số phức
28: };
29:
30: Complex::Complex()
31: {
32: Real = 0.0;
33: Imaginary = 0.0;
34: }
35:
36: Complex::Complex(double R,double I)
37: {
38: Real = R;
39: Imaginary = I;
40: }
41:
42: Complex::Complex(const Complex & Z)
43: {
44: Real = Z.Real;
45: Imaginary = Z.Imaginary;
46: }
47:
48: Complex::Complex(double R)
49: {
50: Real = R;
51: Imaginary = 0.0;
52: }
53:
54: void Complex::Print()
55: {
56: cout<<'('<<Real<<','<<Imaginary<<')';
57: }
58:
59: Complex Complex::operator + (Complex Z)
60: {
61: Complex Tmp;
62
63: Tmp.Real = Real + Z.Real;
64: Tmp.Imaginary = Imaginary + Z.Imaginary;
65: return Tmp;
66: }
67:
68: Complex Complex::operator - (Complex Z)
```

```
69: {
70: Complex Tmp;
71:
72: Tmp.Real = Real - Z.Real;
73: Tmp.Imaginary = Imaginary - Z.Imaginary;
74: return Tmp;
75: }
76:
77: Complex Complex::operator += (Complex Z)
78: {
79: Real += Z.Real;
80: Imaginary += Z.Imaginary;
81: return *this;
82: }
83:
84: Complex Complex::operator -= (Complex Z)
85: {
86: Real -= Z.Real;
87: Imaginary -= Z.Imaginary;
88: return *this;
89: }
90:
91: int Complex::operator == (Complex Z)
92: {
93: return (Real == Z.Real) && (Imaginary == Z.Imaginary);
94: }
95:
96: int Complex::operator != (Complex Z)
97: {
98: return (Real != Z.Real) || (Imaginary != Z.Imaginary);
99: }
100:
101: int Complex::operator > (Complex Z)
102: {
103: return Abs() > Z.Abs();
104: }
105:
106: int Complex::operator >= (Complex Z)
107: {
108: return Abs() >= Z.Abs();
109: }
110:
111: int Complex::operator < (Complex Z)
112: {
113: return Abs() < Z.Abs();
114: }
115:
116: int Complex::operator <= (Complex Z)
117: {
118: return Abs() <= Z.Abs();
119: }
120:
121: double Complex::Abs()
122: {
123: return sqrt(Real*Real+Imaginary*Imaginary);
124: }
125:
126: int main()
```

```
127: {
128: Complex X, Y(4.3,8.2), Z(3.3,1.1), T;
129
130: cout<<"X: ";
131: X.Print();
132: cout<<endl<<"Y: ";
133: Y.Print();
134: cout<<endl<<"Z: ";
135: Z.Print();
136: cout<<endl<<"T: ";
137: T.Print();
138: T=5.3;// Gọi constructor chuyển kiểu
139: cout<<endl<<endl<<"T = 5.3"<<endl;
140: cout<<"T: ";
141: T.Print();
142: X = Y + Z;
143: cout<<endl<<endl<<"X = Y + Z: ";
144: X.Print();
145: cout<<" = ";
146: Y.Print();
147: cout<<" + ";
148: Z.Print();
149: X = Y - Z;
150: cout<<endl<<"X = Y - Z: ";
151: X.Print();
152: cout<<" = ";
153: Y.Print();
154: cout<<" - ";
155: Z.Print();
156: cout<<endl<<endl<<"Y += T i.e ";
157: Y.Print();
158: cout<<" += ";
159: T.Print();
160: Y += T;
161: cout<<endl<<"Y: ";
162: Y.Print();
163: cout<<endl<<"Z -= T i.e ";
164: Z.Print();
165: cout<<" -= ";
166: T.Print();
167: Z -= T;
168: cout<<endl<<"Z: ";
169: Z.Print();
170: Complex U(X);// Gọi constructor sao chép
171: cout<<endl<<endl<<"U: ";
172: U.Print();
173: cout<<endl<<endl<<"Evaluating: X==U"<<endl;
174: if (X==U)
175: cout<<"They are equal"<<endl;
176: cout<<"Evaluating: Y!=Z"<<endl;
177: if (Y!=Z)
178: cout<<"They are not equal => ";
179: if (Y>Z)
180: cout<<"Y>Z";
181: else
182: cout<<"Y<Z";
183: return 0;
184: }
```

Chúng ta chạy ví dụ 4.2, kết quả ở hình 4.6.

Dòng thứ 10 của chương trình ở ví dụ 4.2: Complex(const Complex &Z);

là một constructor sao chép (copy constructor). Nó khởi động một đối tượng lớp bằng cách tạo một sao chép của một đối tượng lớp đó. Constructor sao chép thực hiện công việc giống như toán tử sao chép nhưng nó có một vai trò đặc biệt. Constructor sao chép chỉ nhận tham số là một tham chiếu chỉ đến đối tượng thuộc chính lớp mà nó được định nghĩa. Các constructor sao chép được dùng mỗi khi một sự sao chép của một đối tượng cần thiết như khi có sự truyền tham số bằng trị, khi trả về một đối tượng từ hàm, hoặc khi khởi động một đối tượng mà được sao chép từ đối tượng khác của cùng lớp. Chẳng hạn:

```
Complex A(3.5, 4.5);
Complex B(A); // Gọi constructor sao chép
Complex C = B; // Gọi constructor sao chép
……
Complex MyFunc(Complex Z) // Gọi constructor sao chép
{ rZ; // Gọi constructor sao chép }
```

```
X: (0,0)
Y: (4.3,8.2)
Z: (3.3,1.1)
T: (0,0)

T = 5.3
T: (5.3,0)

X = Y + Z: (7.6,9.3) = (4.3,8.2) + (3.3,1.1)
X = Y - Z: (7.6,9.3) = (4.3,8.2) - (3.3,1.1)

Y += T  i.e  (4.3,8.2) += (5.3,0)
Y: (9.6,8.2)
Z -= T  i.e  (3.3,1.1) -= (5.3,0)
Z: (-2,1.1)

U: (7.6,9.3)

Evaluating: X==U
They are equal
Evaluating: Y!=Z
They are not equal => Y>Z
```

Hình 4.6: Kết quả của ví dụ 4.2

Chúng ta chú ý rằng, dấu = trong câu lệnh trên ứng với constructor sao chép chứ không phải là toán tử gán . Nếu chúng ta không định nghĩa constructor sao chép, trình biên dịch tạo ra một constructor sao chép mặc định sẽ sao chép từng thành viên một.

Ở dòng 12 của chương trình ở ví dụ 4.2:

Complex(double R);

là một constructor chuyển đổi (conversion constructor). Constructor này lấy một tham số **double** và khởi tạo đối tượng *Complex* mà phần thực bằng giá trị tham số truyền vào và phần ảo bằng 0.0 (từ dòng 48 đến 52). Bất kỳ một constructor nào có tham số đơn có thể được nghĩ như một constructor chuyển đổi. Constructor chuyển đổi sẽ đổi một số thực thành một đối tượng *Complex* rồi gán cho đối tượng đích *Complex*. Chẳng hạn:

T = 3.5; // Ngầm định: T = Complex(3.5)

Trình biên dịch tự động dùng constructor chuyển đổi để tạo một đối tượng tạm thời *Complex*, rồi dùng toán tử gán để gán đối tượng tạm thời này cho đối tượng khác của *Complex*. Chẳng hạn câu lệnh sau vẫn đúng:

X = Y + 3.5; // Ngầm định: X = Y + Complex(3.5);

Như vậy một constructor chuyển đổi được sử dụng để thực hiện một sự chuyển đổi ngầm định.

Ví dụ 4.3: Lấy lại ví dụ 4.2 nhưng các hàm toán tử +, - và các hàm toán tử so sánh là hàm không thành viên.

```
#include <iostream.h>
#include <math.h>
class Complex
{
  private:
        double Real,Imaginary;
  public:
        Complex();//Constructor mac dinh
        Complex(double R,double I);
        Complex (const Complex & Z);//Constructor sao chep
        Complex (double R);//Constructor chuyen doi
        void Print();//Hien thi so phuc
        //Cac toan tu tinh toan
        friend Complex operator + (Complex Z1,Complex Z2);
        friend Complex operator - (Complex Z1,Complex Z2);
        Complex operator += (Complex Z);
        Complex operator -= (Complex Z);
        //Cac toan tu so sanh
        friend int operator == (Complex Z1,Complex Z2);
        friend int operator != (Complex Z1,Complex Z2);
        friend int operator > (Complex Z1,Complex Z2);
        friend int operator >= (Complex Z1,Complex Z2);
        friend int operator < (Complex Z1,Complex Z2);
        friend int operator <= (Complex Z1,Complex Z2);
  private:
        double Abs();//Gia tri tuyet doi cua so phuc
};
Complex::Complex()
{
  Real = 0.0;
  Imaginary = 0.0;
}
Complex::Complex(double R,double I)
{
  Real = R;
  Imaginary = I;
}
Complex::Complex(const Complex & Z)
{
  Real = Z.Real;
  Imaginary = Z.Imaginary;
}

Complex::Complex(double R)
{
  Real = R;
  Imaginary = 0.0;
}
void Complex::Print()
```

```
{
   cout<<'('<<Real<<','<<Imaginary<<')';
}
Complex operator + (Complex Z1,Complex Z2)
{
   Complex Tmp;
   Tmp.Real = Z1.Real + Z2.Real;
   Tmp.Imaginary = Z1.Imaginary + Z2.Imaginary;
   return Tmp;
}
Complex operator - (Complex Z1,Complex Z2)
{
   Complex Tmp;
   Tmp.Real = Z1.Real - Z2.Real;
   Tmp.Imaginary = Z1.Imaginary - Z2.Imaginary;
   return Tmp;
}
Complex Complex::operator += (Complex Z)
{
   Real += Z.Real;
   Imaginary += Z.Imaginary;
   return *this;
}
Complex Complex::operator -= (Complex Z)
{
   Real -= Z.Real;
   Imaginary -= Z.Imaginary;
   return *this;
}
int operator == (Complex Z1,Complex Z2)
{
   return (Z1.Real == Z2.Real) && (Z1.Imaginary == Z2.Imaginary);
}
int operator != (Complex Z1,Complex Z2)
{
   return (Z1.Real != Z2.Real) || (Z1.Imaginary != Z2.Imaginary);
}
int operator > (Complex Z1,Complex Z2)
{
   return Z1.Abs() > Z2.Abs();
}
int operator >= (Complex Z1,Complex Z2)
{
   return Z1.Abs() >= Z2.Abs();
}
int operator < (Complex Z1,Complex Z2)
{
   return Z1.Abs() < Z2.Abs();
}
int operator <= (Complex Z1,Complex Z2)
{
   return Z1.Abs() <= Z2.Abs();
}
double Complex::Abs()
{
   return sqrt(Real*Real+Imaginary*Imaginary);
}
int main()
```

```
{
   Complex X,Y(4.3,8.2),Z(3.3,1.1);
   cout<<"X: ";       X.Print();
   cout<<endl<<"Y: "; Y.Print();
   cout<<endl<<"Z: "; Z.Print();
   X = Y + 3.6;
   cout<<endl<<endl<<"X = Y + 3.6: ";
   X.Print();       cout<<" = ";
   Y.Print();       cout<<" + 3.6 ";
   X = 3.6 + Y;     cout<<endl<<"X = 3.6 + Y: ";
   X.Print();       cout<<" = 3.6 + ";
   Y.Print();       X = 3.8 - Z;
   cout<<endl<<"X = 3.8 - Z: ";
   X.Print();       cout<<" = 3.8 - ";
   Z.Print();       X = Z - 3.8;
   cout<<endl<<"X = Z - 3.8: ";
   X.Print();       cout<<" = ";
   Z.Print();       cout<<" - 3.8 ";
   return 0;
}
```

Chúng ta chạy ví dụ 4.3, kết quả ở hình 4.7

```
X: (0,0)
Y: (4.3,8.2)
Z: (3.3,1.1)

X = Y + 3.6: (7.9,8.2) = (4.3,8.2) + 3.6
X = 3.6 + Y: (7.9,8.2) = 3.6 + (4.3,8.2)
X = 3.8 - Z: (0.5,-1.1) = 3.8 - (3.3,1.1)
X = Z - 3.8: (-0.5,1.1) = (3.3,1.1) - 3.8
```

Hình 4.7: Kết quả của ví dụ 4.3

VI. ĐA NĂNG HÓA CÁC TOÁN TỬ MỘT NGÔI

Các toán tử một ngôi được đa năng hóa trong hình 4.8 sau:

Toán tử	Ví dụ	Toán tử	Ví dụ
+	+c	~	~c
-	-c	!	!a
*	*c	++	++c, c++
&	&c	--	--c, c--
->	c->		

Hình 4.8: Các toán tử một ngôi được đa năng hóa

Một toán tử một ngôi của lớp được đa năng hóa như một hàm thành viên không tĩnh với không có tham số hoặc như một hàm không thành viên với một tham số; Tham số đó phải hoặc là một đối tượng lớp hoặc là một tham chiếu đến đối tượng lớp.

Ví dụ 4.4: Lấy lại ví dụ 4.3 và thêm toán tử dấu trừ một ngôi.

```
1: #include <iostream.h>
2: #include <math.h>
3:
4: class Complex
5: {
```

```
6: private:
7: double Real,Imaginary;
8: public:
9: Complex(); // Constructor mặc định
10: Complex(double R,double I);
11: Complex (const Complex & Z); // Constructor sao chép
12: Complex (double R); // Constructor chuyển đổi
13: void Print(); // Hiển thị số phức
14: // Các toán tử tính toán
15: friend Complex operator + (Complex Z1,Complex Z2);
16: friend Complex operator - (Complex Z1,Complex Z2);
17: Complex operator += (Complex Z);
18: Complex operator -= (Complex Z);
19: // Toán tử trừ một ngôi
20: Complex operator - ();
21: // Các toán tử so sánh
22: friend int operator == (Complex Z1,Complex Z2);
23: friend int operator != (Complex Z1,Complex Z2);
24: friend int operator > (Complex Z1,Complex Z2);
25: friend int operator >= (Complex Z1,Complex Z2);
26: friend int operator < (Complex Z1,Complex Z2);
27: friend int operator <= (Complex Z1,Complex Z2);
28: private:
29: double Abs(); // Giá trị tuyệt đối của số phức
30: };
31:
32: Complex::Complex()
33: {
34: Real = 0.0;
35: Imaginary = 0.0;
36: }
37:
38: Complex::Complex(double R,double I)
39: {
40: Real = R;
41: Imaginary = I;
42: }
43:
44: Complex::Complex(const Complex & Z)
45: {
46: Real = Z.Real;
47: Imaginary = Z.Imaginary;
48: }
49:
50: Complex::Complex(double R)
51: {
52: Real = R;
53: Imaginary = 0.0;
54: }
55:
56: void Complex::Print()
57: {
58: cout<<'('<<Real<<','<<Imaginary<<')';
59: }
60:
61: Complex operator + (Complex Z1,Complex Z2)
62: {
63: Complex Tmp;
```

```
64:
65: Tmp.Real = Z1.Real + Z2.Real;
66: Tmp.Imaginary = Z1.Imaginary + Z2.Imaginary;
67: return Tmp;
68: }
69:
70: Complex operator - (Complex Z1,Complex Z2)
71: {
72: Complex Tmp;
73:
74: Tmp.Real = Z1.Real - Z2.Real;
75: Tmp.Imaginary = Z1.Imaginary - Z2.Imaginary;
76: return Tmp;
77: }
78:
79: Complex Complex::operator += (Complex Z)
80: {
81: Real += Z.Real;
82: Imaginary += Z.Imaginary;
83: return *this;
84: }
85:
86: Complex Complex::operator -= (Complex Z)
87: {
88: Real -= Z.Real;
89: Imaginary -= Z.Imaginary;
90: return *this;
91: }
92:
93: Complex Complex::operator - ()
94: {
95: Complex Tmp;
96:
97: Tmp.Real = -Real;
98: Tmp.Imaginary = -Imaginary;
99: return Tmp;
100: }
101
102: int operator == (Complex Z1,Complex Z2)
103: {
104: return (Z1.Real == Z2.Real) && (Z1.Imaginary == Z2.Imaginary);
105: }
106:
107: int operator != (Complex Z1,Complex Z2)
108: {
109: return (Z1.Real != Z2.Real) || (Z1.Imaginary != Z2.Imaginary);
110: }
111:
112: int operator > (Complex Z1,Complex Z2)
113: {
114: return Z1.Abs() > Z2.Abs();
115: }
116:
117: int operator >= (Complex Z1,Complex Z2)
118: {
119: return Z1.Abs() >= Z2.Abs();
120: }
121:
```

```
122: int operator < (Complex Z1,Complex Z2)
123: {
124: return Z1.Abs() < Z2.Abs();
125: }
126:
127: int operator <= (Complex Z1,Complex Z2)
128: {
129: return Z1.Abs() <= Z2.Abs();
130: }
131:
132: double Complex::Abs()
133: {
134: return sqrt(Real*Real+Imaginary*Imaginary);
135: }
136:
137: int main()
138: {
139: Complex X, Y(4.3,8.2), Z(3.3,1.1);
140:
141: cout<<"X: ";
142: X.Print();
143: cout<<endl<<"Y: ";
144: Y.Print();
145: cout<<endl<<"Z: ";
146: Z.Print();
147: X = -Y + 3.6;
148: cout<<endl<<endl<<"X = -Y + 3.6: ";
149: X.Print();
150: cout<<" = ";
151: (-Y).Print();
152: cout<<" + 3.6 ";
153: X = -Y + -Z;
154: cout<<endl<<"X = -Y + -Z: ";
155: X.Print();
156: cout<<" = ";
157: (-Y).Print();
158: cout<<" + ";
159: (-Z).Print();
160: return 0;
161: }
```

Chúng ta chạy ví dụ 4.4, kết quả ở hình 4.9

```
X: (0,0)
Y: (4.3,8.2)
Z: (3.3,1.1)

X = -Y + 3.6: (-0.7,-8.2) = (-4.3,-8.2) + 3.6
X = -Y + -Z: (-7.6,-9.3) = (-4.3,-8.2) + (-3.3,-1.1)
```

Hình 4.9: Kết quả của ví dụ 4.4

VII. ĐA NĂNG HÓA MỘT SỐ TOÁN TỬ ĐẶC BIỆT

Trong phần này chúng ta sẽ tìm hiểu cách cài đặt một vài toán tử đặc biệt như **() [] ++ -- , = ->**

VII.1. Toán tử []

Khi cài đặt các lớp vector hoặc chuỗi ký tự, chúng ta cần phải truy cập đến từng phần tử của chúng, trong ngôn ngữ C/C++ đã có toán tử **[]** để truy cập đến một phần tử của mảng. Đây là toán tử hai ngôi, có dạng a[b] và khi đa năng toán tử này thì hàm toán tử tương ứng phải là thành viên của một lớp.

Ví dụ 4.5: Đa năng hóa toán tử **[]** để truy cập đến một phần tử của vector.

```
1: #include <iostream.h>
2:
3: class Vector
4: {
5: private:
6: int Size;
7: int *Data;
8: public:
9: Vector(int S=2,int V=0);
10: ~Vector();
11: void Print() const;
12: int & operator [] (int I);
13: };
14:
15: Vector::Vector(int S,int V)
16: {
17: Size = S;
18: Data=new int[Size];
19: for(int I=0;I<Size;++I)
20: Data[I]=V;
21: }
22:
23: Vector::~Vector()
24: {
25: delete []Data;
26: }
27: void Vector::Print() const
28: {
29: cout<<"Vector:(";
30: for(int I=0;I<Size-1;++I)
31: cout<<Data[I]<<",";
32: cout<<Data[Size-1]<<")"<<endl;
33: }
34:
35: int & Vector::operator [](int I)
36: {
37: return Data[I];
38: }
39:
40: int main()
41: {
42: Vector V(5,1);
43: V.Print();
44: for(int I=0;I<5;++I)
45: V[I]*=(I+1);
46: V.Print();
47: V[0]=10;
48: V.Print();
49: return 0;
50: }
```

Chúng ta chạy ví dụ 4.5, kết quả ở hình 4.10

```
Vector:(1,1,1,1,1)
Vector:(1,2,3,4,5)
Vector:(10,2,3,4,5)
```

Hình 4.10: Kết quả của ví dụ 4.5

Trong chương trình ở ví dụ 4.5, hàm toán tử của toán tử **[]** ở lớp *Vector* trả về một tham chiếu vì toán tử này có thể dùng ở vế trái của phép gán.

VII.2. Toán tử ()

Toán tử **()** được dùng để gọi hàm, toán tử này gồm hai toán hạng: toán hạng đầu tiên là tên hàm, toán hạng thứ hai là danh sách các tham số của hàm. Toán tử này có dạng giống như toán tử **[]** và khi đa năng toán tử này thì hàm toán tử tương ứng phải là thành viên của một lớp.

Ví dụ 4.6: Lấy lại ví dụ 4.5 nhưng đa năng hóa toán tử **()** để truy cập đến một phần tử của vector.

```
1: #include <iostream.h>
2:
3: class Vector
4: {
5: private:
6: int Size;
7: int *Data;
8: public:
9: Vector(int S=2,int V=0);
10: ~Vector();
11: void Print() const;
12: int & operator () (int I);
13: };
14:
15: Vector::Vector(int S,int V)
16: {
17: Size = S;
18: Data=new int[Size];
19: for(int I=0;I<Size;++I)
20: Data[I]=V;
21: }
22:
23: Vector::~Vector()
24: {
25: delete []Data;
26: }
27: void Vector::Print() const
28: {
29: cout<<"Vector:(";
30: for(int I=0;I<Size-1;++I)
31: cout<<Data[I]<<",";
32: cout<<Data[Size-1]<<")"<<endl;
33: }
34:
35: int & Vector::operator ()(int I)
36: {
37: return Data[I];
38: }
39:
40: int main()
41: {
42: Vector V(5,1);
```

```
43: V.Print();
44: for(int I=0;I<5;++I)
45: V(I)*=(I+1);
46: V.Print();
47: V(0)=10;
48: V.Print();
49: return 0;
50: }
```

Chúng ta chạy ví dụ 4.6, kết quả ở hình 4.11

```
Vector:(1,1,1,1,1)
Vector:(1,2,3,4,5)
Vector:(10,2,3,4,5)
```

Hình 4.11: Kết quả của ví dụ 4.6

Ví dụ 4.7: Đa năng hóa toán tử **()** để truy cập đến phần tử của ma trận.

```
1: #include <iostream.h>
2:
3: class Matrix
4: {
5: private:
6: int Rows,Cols;
7: int **Data;
8: public:
9: Matrix(int R=2,int C=2,int V=0);
10: ~Matrix();
11: void Print() const;
12: int & operator () (int R,int C);
13: };
14:
15: Matrix::Matrix(int R,int C,int V)
16: {
17: int I,J;
18: Rows=R;
19: Cols=C;
20: Data = new int *[Rows];
21: int *Temp=new int[Rows*Cols];
22: for(I=0;I<Rows;++I)
23: {
24: Data[I]=Temp;
25: Temp+=Cols;
26: }
27: for(I=0;I<Rows;++I)
28: for(J=0;J<Cols;++J)
29: Data[I][J]=V;
30: }
31:
32: Matrix::~Matrix()
33: {
34: delete [] Data[0];
35: delete [] Data;
36: }
37:
38: void Matrix::Print() const
39: {
40: int I,J;
41: for(I=0;I<Rows;++I)
```

```
42: {
43: for(J=0;J<Cols;++J)
44: {
45: cout.width(5); // Hiển thị canh lề phải với chiều dài 5 ký tự
46: cout<<Data[I][J];
47: }
48: cout<<endl;
49: }
50: }
51:
52: int & Matrix::operator () (int R,int C)
53: {
54: return Data[R][C];
55: }
56:
57: int main()
58: {
59: int I,J;
60: Matrix M(2,3,1);
61: cout<<"Matrix:"<<endl;
62: M.Print();
63: for(I=0;I<2;++I)
64: for(J=0;J<3;++J)
65: M(I,J)*=(I+J+1);
66: cout<<"Matrix:"<<endl;
67: M.Print();
68: return 0;
69: }
```

Chúng ta chạy ví dụ 4.7, kết quả ở hình 4.12

```
Matrix:
  1  1  1
  1  1  1
Matrix:
  1  2  3
  2  3  4
```

Hình 4.12: Kết quả của ví dụ 4.7

VIII. TOÁN TỬ CHUYỂN ĐỔI KIỂU

Phần lớn các chương trình xử lý thông tin sự đa dạng của các kiểu. Đôi khi tất cả các thao tác "dừng lại bên trong một kiểu". Chẳng hạn, một số nguyên với một số nguyên tạo thành một số nguyên (miễn là kết quả không quá lớn để được biểu diễn như một số nguyên). Nhưng thật cần thiết để chuyển đổi dữ liệu của một kiểu tới dữ liệu của kiểu khác. Điều này có thể xảy ra trong các phép gán, các kết quả tính toán, trong việc chuyển các giá trị tới hàm, và trong việc trả về trị từ hàm. Trình biên dịch biết làm thế nào để thực hiện các chuyển đổi nào đó trong số các kiểu có sẵn. Các lập trình viên có thể ép buộc các chuyển đổi trong số các kiểu có sẵn bởi ép kiểu.

Nhưng đối với các kiểu do người dùng định nghĩa thì trình biên dịch không thể tự động biết làm thế nào chuyển đổi trong số các kiểu dữ liệu do người dùng định nghĩa và các kiểu có sẵn. Lập trình viên phải chỉ rõ làm sao các chuyển đổi như vậy sẽ xuất hiện. Các chuyển đổi như thế có thể được thực hiện với constructor chuyển đổi.

Một toán tử chuyển đổi kiểu có thể được sử dụng để chuyển đổi một đối tượng của một lớp thành đối tượng của một lớp khác hoặc thành một đối tượng của một kiểu có sẵn. Toán tử chuyển đổi kiểu như thế phải là hàm thành viên không tĩnh và không là hàm **friend**. Prototype của hàm thành viên này có cú pháp:

operator <*data type*> **();**

Ví dụ 4.14: Toán tử chuyển đổi kiểu

```
1: #include <iostream.h>
2:
3: class Number
4: {
5: private:
6: float Data;
7: public:
8: Number(float F=0.0)
9: {
10: Data=F;
11: }
12: operator float()
13: {
14: return Data;
15: }
16: operator int()
17: {
18: return (int)Data;
19: }
20: };
21:
22: int main()
23: {
24: Number N1(9.7), N2(2.6);
25: float X=(float)N1; //Gọi operator float()
26: cout<<X<<endl;
27: int Y=(int)N2; //Gọi operator int()
28: cout<<Y<<endl;
29: return 0;
30: }
```

Chúng ta chạy ví dụ 4.14, kết quả ở hình 4.19

```
9.7
2
```

Hình 4.19: Kết quả của ví dụ 4.14

IX. TOÁN TỬ NEW VÀ DELETE

Các toán tử **new** và **delete** toàn cục có thể được đa năng hóa. Điều này cho phép các lập trình viên C++ có khả năng xây dựng một hệ thống cấp phát bộ nhớ theo ý người dùng, cói cùng giao tiếp như hệ thống cấp phát mặc định.

Có hai cách đa năng hóa các toán tử **new** và **delete**:

- Có thể đa năng hóa một cách toàn cục nghĩa là thay thế hẳn các toán tử **new** và **delete** mặc định.
- Chúng ta đa năng hóa các toán tử **new** và **delete** với tư cách là hàm thành viên của lớp nếu muốn các toán tử **new** và **delete** áp dụng đối với lớp đó. Khi chúng ta dùng **new** và **delete** đối với lớp nào đó, trình biên dịch sẽ kiểm tra xem **new** và **delete** có được định nghĩa riêng cho lớp đó hay không; nếu không thì dùng **new** và **delete** toàn cục (có thể đã được đa năng hóa).

Hàm toán tử của toán tử **new** và **delete** có prototype như sau:

void * operator new(size_t *size***);**

void operator delete(void * *ptr***);**

Trong đó tham số kiểu **size_t** được trình biên dịch hiểu là kích thước của kiểu dữ liệu được trao cho toán tử **new**.

IX.1. Đa năng hóa toán tử new và delete toàn cục

Ví dụ 4.15: Đa năng hóa toán tử **new** và **delete** toàn cục đồng thời chứng tỏ rằng toán tử **new** và **delete** do đa năng hóa thay thế toán tử **new** và **delete** mặc định.

```
1: #include <iostream.h>
2: #include <stdlib.h>
3:
4: class Point
5: {
6: private:
7: int X, Y;
8: public:
9: Point(int A=0,int B=0)
10: {
11: X=A;
12: Y=B;
13: cout<<"Constructor!"<<endl;
14: }
15: ~Point()
16: {
17: cout<<"Destructor!"<<endl;
18: }
19: void Print() const
20: {
21: cout<<"X="<<X<<","<<"Y="<<Y<<endl;
22: }
23: };
24:
25: void * operator new(size_t Size)
26: {
27: return malloc(Size);
28: }
29:
30: void operator delete(void *Ptr)
31: {
32: free(Ptr);
33: }
34:
35: int main()
36: {
37: Point *P1,*P2;
38: P1= new Point(10,20);
39: if (P1==NULL)
40: {
41: cout<<"Out of memory!"<<endl;
42: return 1;
43: }
44: P2= new Point(-10,-20);
45: if (P2==NULL)
46: {
47: cout<<"Out of memory!"<<endl;
48: return 1;
49: }
50: int *X=new int;
51: if (X==NULL)
52: {
53: cout<<"Out of memory!"<<endl;
54: return 1;
```

```
55: }
56: *X=10;
57: cout<<"X="<<*X<<endl;
58: cout<<"Point 1:";
59: P1->Print();
60: cout<<"Point 2:";
61: P2->Print();
62: delete P1;
63: delete P2;
64: delete X;
65: return 0;
66: }
```

Chúng ta chạy ví dụ 4.15, kết quả ở hình 4.20

```
Constructor!
Constructor!
X=10
Point 1:X=10,Y=20
Point 2:X=-10,Y=-20
Destructor!
Destructor!
```

Hình 4.20: Kết quả của ví dụ 4.15

IX.2. Đa năng hóa toán tử new và delete cho một lớp

Nếu muốn toán tử **new** và **delete** có tính chất đặc biệt chỉ khi áp dụng cho đối tượng của lớp nào đó, chúng ta có thể đa năng hóa toán tử **new** và **delete** với tư cách là hàm thành viên của lớp đó. Việc này không khác lắm so với cách đa năng hóa toán tử **new** và **delete** một cách toàn cục.

Ví dụ 4.16: Đa năng hóa toán tử **new** và **delete** cho một lớp.

```
1: #include <iostream.h>
2: #include <stdlib.h>
3: class Number
4: {
5: private:
6: int Data;
7: public:
8: Number(int X=0)
9: {
10: Data=X;
11: }
12:
13: void * operator new(size_t Size)
14: {
15: cout<<"Toan tu new cua lop!"<<endl;
16: return ::new unsigned char[Size];
17: }
18:
19: void operator delete(void *Ptr)
20: {
21: cout<<"Toan tu delete cua lop!"<<endl;
22: ::delete Ptr;
23: }
24:
25: void Print() const
26: {
27: cout<<"Data:"<<Data<<endl;
28: }
```

```
29:
30: };
31:
32: int main()
33: {
34: Number *N;
35: N= new Number(10);
36: if (N==NULL)
37: {
38: cout<<"Out of memory!"<<endl;
39: return 1;
40: }
41: int *X=new int;
42: if (X==NULL)
43: {
44: cout<<"Out of memory!"<<endl;
45: return 1;
46: }
47: *X=10;
48: cout<<"X="<<*X<<endl;
49: N->Print();
50: delete N;
51: delete X;
52: return 0;
53: }
```

Chúng ta chạy ví dụ 4.16, kết quả ở hình 4.21

```
Toan tu new cua lop!
X=10
Data: 10
Toan tu delete cua lop!
```

Hình 4.21: Kết quả của ví dụ 4.16

X. ĐA NĂNG HÓA CÁC TOÁN TỬ CHÈN DÒNG << VÀ TRÍCH DÒNG >>

Chúng ta có thể đa năng hóa các toán tử chèn dòng << (stream insertion) và trích dòng >> (stream extraction). Hàm toán tử của toán tử << được đa năng hóa có prototype như sau:

ostream & operator << (ostream & *stream***,** *ClassName Object***);**

Hàm toán tử << trả về tham chiếu chỉ đến dòng xuất **ostream**. Tham số thứ nhất của hàm toán tử << là một tham chiếu chỉ đến dòng xuất **ostream**, tham số thứ hai là đối tượng được chèn vào dòng. Khi sử dụng, dòng trao cho toán tử << (tham số thứ nhất) là toán hạng bên trái và đối tượng được đưa vào dòng (tham số thứ hai) là toán hạng bên phải. Để bảo đảm cách dùng toán tử << luôn nhất quán, chúng ta không thể định nghĩa hàm toán tử << như là hàm thành viên của lớp đang xét, thông thường nó chính là hàm **friend**.

Còn hàm toán tử của toán tử >> được đa năng hóa có prototype như sau:

istream & operator >> (istream & *stream***,** *ClassName Object***);**

Hàm toán tử >> trả về tham chiếu chỉ đến dòng nhập istream. Tham số thứ nhất của hàm toán tử này là một tham chiếu chỉ đến dòng nhập istream, tham số thứ hai là đối tượng của lớp đang xét mà chúng ta muốn tạo dựng nhờ vào dữ liệu lấy từ dòng nhập. Khi sử dụng, dòng nhập đóng vai toán hạng bên trái, đối tượng nhận dữ liệu đóng vai toán hạng bên phải. Cũng như trường hợp toán tử <<, hàm toán tử >> không là hàm thành viên của lớp, thông thường nó chính là hàm **friend**.

Ví dụ 4.17:

```
1: #include <iostream.h>
2:
3: class Point
4: {
5: private:
6: int X,Y;
7: public:
8: Point();
9: friend ostream & operator << (ostream & Out,Point & P);
10: friend istream & operator >> (istream & In,Point & P);
11: };
12:
13: Point::Point()
14: {
15: X=Y=0;
16: }
17:
18: ostream & operator << (ostream & Out,Point & P)
19: {
20: Out<<"X="<<P.X<<",Y="<<P.Y<<endl;
21: return Out; //Cho phép cout<<a<<b<<c;
22: }
23:
24: istream & operator >> (istream &In,Point & P)
25: {
26: cout<<"X:";
27: In>>P.X;
28: cout<<"Y:";
29: In>>P.Y;
30: return In; //Cho phép cin>>a>>b>>c;
31: }
32:
33: int main()
34: {
35: Point P;
36: cin>>P;
37: cout<<"Point:"<<P;
38: return 0;
39: }
```

Chúng ta chạy ví dụ 4.17, kết quả ở hình 4.22

```
X:1
Y:3
Point:X=1,Y=3
```

Hình 4.22: Kết quả của ví dụ 4.17

XI. MỘT SỐ VÍ DỤ

XI.1. Lớp String

Ví dụ 4.18: Chúng ta sẽ xây dựng một lớp xử lý việc tạo và thao tác trên các chuỗi (string). C++ không cài sẵn kiểu dữ liệu chuỗi. Nhưng C++ cho phép chúng ta thêm kiểu chuỗi như một lớp thông qua cơ chế đa năng hóa.

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
```

```
#include <assert.h>
class String
{
      private:
            char *Ptr;   //Con tro tro den diem bat dau cua chuoi
            int Length;  //Chieu dai chuoi
      public:
            String(const char * = ""); //Constructor chuyen doi
            String(const String &);    //Constructor sao chep
            ~String();                 //Destructor
            const String &operator=(const String &);  //Phep gan
            String &operator+=(const String &);       //Phep noi
            int operator!() const;                    //Kiem tra chuoi rong
            int operator==(const String &) const;
            int operator!=(const String &) const;
            int operator<(const String &)  const;
            int operator>(const String &)  const;
            int operator>=(const String &) const;
            int operator<=(const String &) const;
            char & operator[](int);       //Tra ve ky tu tham chieu
            String &operator()(int, int); //Tra ve mot chuoi con
            int GetLength() const;
            friend ostream &operator<<(ostream &, const String &);
            friend istream &operator>>(istream &, String &);
};
//Constructor sao chep: Chuyen doi char * thanh String
String::String(const char *S)
{
      cout << "Conversion constructor: " << S << endl;
      Length = strlen(S);
      Ptr = new char[Length + 1];
      assert(Ptr != 0);

      strcpy(Ptr, S);
}
String::String(const String &Copy)
{
      cout << "Copy constructor: " << Copy.Ptr << endl;
      Length = Copy.Length;
      Ptr = new char[Length + 1];
      assert(Ptr != 0);
      strcpy(Ptr, Copy.Ptr);
}
//Destructor
String::~String()
{
      cout << "Destructor: " << Ptr << endl;
      delete [] Ptr;
}
const String &String::operator=(const String &Right)
{
      cout << "operator= called" << endl;
      if (&Right != this)
      {
            delete [] Ptr;
            Length = Right.Length;
            Ptr = new char[Length + 1];
            assert(Ptr != 0);
            strcpy(Ptr, Right.Ptr);
      }
      else
            cout << "Attempted assignment of a String to itself" << endl;
      return *this;
}
```

```
String &String::operator+=(const String &Right)
{
      char *TempPtr = Ptr;
      Length += Right.Length;
      Ptr = new char[Length + 1];
      assert(Ptr != 0);
      strcpy(Ptr, TempPtr);
      strcat(Ptr, Right.Ptr);
      delete [] TempPtr;
      return *this;
}
int String::operator!() const
{
      return Length == 0;
}
int String::operator==(const String &Right) const
{
      return strcmp(Ptr, Right.Ptr) == 0;
}
int String::operator!=(const String &Right) const
{
      return strcmp(Ptr, Right.Ptr) != 0;
}
int String::operator<(const String &Right) const
{
      return strcmp(Ptr, Right.Ptr) < 0;
}
int String::operator>(const String &Right) const
{
      return strcmp(Ptr, Right.Ptr) > 0;
}

int String::operator>=(const String &Right) const
{
      return strcmp(Ptr, Right.Ptr) >= 0;
}
int String::operator<=(const String &Right) const
{
      return strcmp(Ptr, Right.Ptr) <= 0;
}
char &String::operator[](int Subscript)
{
      assert(Subscript >= 0 && Subscript < Length);
      return Ptr[Subscript];
}
String &String::operator()(int Index, int SubLength)
{
      assert(Index >= 0 && Index < Length && SubLength >= 0);
      String *SubPtr = new String;
      assert(SubPtr != 0);
      if ((SubLength == 0) || (Index + SubLength > Length))
            SubPtr->Length = Length - Index + 1;
      else
            SubPtr->Length = SubLength + 1;
      delete SubPtr->Ptr;
      SubPtr->Ptr = new char[SubPtr->Length];
      assert(SubPtr->Ptr != 0);
      strncpy(SubPtr->Ptr, &Ptr[Index], SubPtr->Length);
      SubPtr->Ptr[SubPtr->Length] = '\0';
      return *SubPtr;
}
int String::GetLength() const
{     return Length;
```

```
}
ostream &operator<<(ostream &Output, const String &S)
{
      Output << S.Ptr;
      return Output;
}
istream &operator>>(istream &Input, String &S)
{
      char Temp[100];
      Input >> setw(100) >> Temp;
      S = Temp;
      return Input;
}
int main()
{
      String S1("happy"), S2(" birthday"), S3;
      cout << "S1 is \"" << S1 << "\"; S2 is \"" << S2
              << "\"; S3 is \"" << S3 << '\"' << endl
              << "The results of comparing S2 and S1:" << endl
              << "S2 == S1 yields " << (S2 == S1) << endl
              << "S2 != S1 yields " << (S2 != S1) << endl
              << "S2 >  S1 yields " << (S2 > S1) << endl
              << "S2 <  S1 yields " << (S2 < S1) << endl
              << "S2 >= S1 yields " << (S2 >= S1) << endl
              << "S2 <= S1 yields " << (S2 <= S1) << endl;
      cout << "Testing !S3:" << endl;
      if (!S3)
      {
            cout << "S3 is empty; assigning S1 to S3;" << endl;
            S3 = S1;
            cout << "S3 is \"" << S3 << "\"" << endl;
      }
      cout << "S1 += S2 yields S1 = ";
      S1 += S2;
      cout << S1 << endl;
      cout << "S1 += \" to you\" yields" << endl;
      S1 += " to you";
      cout << "S1 = " << S1 << endl;
      cout << "The substring of S1 starting at" << endl
              << "location 0 for 14 characters, S1(0, 14), is: "
              << S1(0, 14) << endl;
      cout << "The substring of S1 starting at" << endl
              << "location 15, S1(15, 0), is: "
              << S1(15, 0) <<endl;  // 0 is "to end of string"
      String *S4Ptr = new String(S1);
      cout << "*S4Ptr = " << *S4Ptr <<endl;
      cout << "assigning *S4Ptr to *S4Ptr" << endl;
      *S4Ptr = *S4Ptr;
      cout << "*S4Ptr = " << *S4Ptr << endl;
      delete S4Ptr;
      S1[0] = 'H';
      S1[6] = 'B';
      cout <<"S1 after S1[0] = 'H' and S1[6] = 'B' is: "<< S1 << endl;
      cout << "Attempt to assign 'd' to S1[30] yields:" << endl;
      S1[30] = 'd'; //Loi: Chi so vuot khoi mien!!!
      return 0;
}
```

Chúng ta chạy ví dụ 4.18, kết quả ở hình 4.23

```
Conversion constructor: happy
Conversion constructor:  birthday
Conversion constructor:
S1 is "happy"; S2 is " birthday"; S3 is "
The results of comparing S2 and S1:
S2 == S1 yields 0
S2 != S1 yields 1
S2 > S1 yields 0
S2 < S1 yields 1
S2 >= S1 yields 0
S2 <= S1 yields 1
Testing !S3:
S3 is empty; assigning S1 to S3;
operator= called
S3 is "happy"
S1 += S2 yields S1 = happy birthday
S1 += " to you" yields
Conversion constructor:  to you
Destructor:  to you
S1 = happy birthday to you
Conversion constructor:
The substring of S1 starting at
location 0 for 14 characters, S1(0, 14), is: happy birthday
Conversion constructor:
The substring of S1 starting at
location 15, S1(15, 0), is: to you
Copy constructor: happy birthday to you
*S4Ptr = happy birthday to you
assigning *S4Ptr to *S4Ptr
operator= called
Attempted assignment of a String to itself
*S4Ptr = happy birthday to you
Destructor: happy birthday to you
S1 after S1[0] = 'H' and S1[6] = 'B' is: Happy Birthday to you
Attempt to assign 'd' to S1[30] yields:
Assertion failed: Subscript >= 0 && Subscript < Length, file CT4_18.CPP, line 12
5
```

Hình 4.23: Kết quả của ví dụ 4.18

XI.2. Lớp Date

Ví dụ 4.19:

```
#include <iostream.h>
class Date
{
      private:
            int Month;
            int Day;
            int Year;
            static int Days[];     //Mang chua so ngay trong thang
            void HelpIncrement();  //Ham tang ngay len mot
      public:
            Date(int M = 1, int D = 1, int Y = 1900);
```

```
            void SetDate(int, int, int);
            Date operator++();            //Tien to
            Date operator++(int);         //Hau to
            const Date &operator+=(int);
            int LeapYear(int);            //Kiem tra nam nhuan
            int EndOfMonth(int);          //Kiem tra cuoi thang
            friend ostream &operator<<(ostream &, const Date &);
};
int Date::Days[] = {31, 28, 31, 30, 31, 30,31, 31, 30, 31, 30, 31};
Date::Date(int M, int D, int Y)
{
      SetDate(M, D, Y);
}
void Date::SetDate(int MM, int DD, int YY)
{
      Month = (MM >= 1 && MM <= 12) ? MM : 1;
      Year = (YY >= 1900 && YY <= 2100) ? YY : 1900;
      if (Month == 2 && LeapYear(Year))
            Day = (DD >= 1 && DD <= 29) ? DD : 1;
      else
            Day = (DD >= 1 && DD <= Days[Month-1]) ? DD : 1;
}
Date Date::operator++()
{
      HelpIncrement();
      return *this;
}
Date Date::operator++(int)
{
      Date Temp = *this;
      HelpIncrement();
      return Temp;
}
const Date &Date::operator+=(int AdditionalDays)
{
      for (int I = 1; I <= AdditionalDays; I++)
            HelpIncrement();

      return *this;
}
int Date::LeapYear(int Y)
{
      if (Y % 400 == 0 || (Y % 100 != 0 && Y % 4 == 0) )
            return 1;  //Nam nhuan
      return 0;     //Nam khong nhuan
}
int Date::EndOfMonth(int D)
{
      if (Month == 2 && LeapYear(Year))
            return D == 29;

      return D == Days[Month-1];
}
void Date::HelpIncrement()
{
      if (EndOfMonth(Day) && Month == 12) //Het nam
      {
            Day = 1;
            Month = 1;
            ++Year;
      }
      else
            if (EndOfMonth(Day)) //Het thang
            {
```

```
                Day = 1;
                ++Month;
            }
            else
                ++Day;
}
ostream &operator<<(ostream &Output, const Date &D)
{
      static char*MonthName[12]={"January","February","March","April","May",
                        "June","July", "August","September",
                        "October","November", "December"  };

      Output << MonthName[D.Month-1] << ' '<< D.Day << ", " << D.Year;
      return Output;
}
int main()
{
      Date D1, D2(12, 27, 1992), D3(0, 99, 8045);
      cout << "D1 is " << D1 << endl
           << "D2 is " << D2 << endl
           << "D3 is " << D3 << endl << endl;
      cout << "D2 += 7 is " << (D2 += 7) << endl << endl;
      D3.SetDate(2, 28, 1992);
      cout << "  D3 is " << D3 << endl;
      cout << "++D3 is " << ++D3 << endl << endl;
      Date D4(3, 18, 1969);
      cout << "Testing the preincrement operator:" << endl
           << "  D4 is " << D4 << endl;
      cout << "++D4 is " << ++D4 << endl;
      cout << "  D4 is " << D4 << endl << endl;
      cout << "Testing the postincrement operator:" << endl
           << "  D4 is " << D4 << endl;
      cout << "D4++ is " << D4++ << endl;
      cout << "  D4 is " << D4 << endl;
      return 0;
}
```

Chúng ta chạy ví dụ 4.19, kết quả ở hình 4.24

```
D1 is January 1, 1900
D2 is December 27, 1992
D3 is January 1, 1900

D2 += 7 is January 3, 1993

 D3 is February 28, 1992
++D3 is February 29, 1992

Testing the preincrement operator:
 D4 is March 18, 1969
++D4 is March 19, 1969
 D4 is March 19, 1969

Testing the postincrement operator:
 D4 is March 19, 1969
D4++ is March 19, 1969
 D4 is March 20, 1969
```

Hình 4.24: Kết quả của ví dụ 4.19

BÀI TẬP

Bài 1: Xây dựng lớp Complex chứa các số phức gồm các phép toán: +, -, *, /, +=, -=, *=, /=, ==, !=, >, >=, <, <=.

Bài 2: Xây dựng lớp String để thực hiện các thao tác trên các chuỗi, trong lớp này có các phép toán:

- Phép toán + để nối hai chuỗi lại với nhau.
- Phép toán = để gán một chuỗi cho một chuỗi khác.
- Phép toán [] truy cập đến một ký tự trong chuỗi.
- Các phép toán so sánh: ==, !=, >, >=, <, <=

Bài 3: Xây dựng lớp ma trận Matrix gồm các phép toán cộng, trừ và nhân hai ma trận bất kỳ.

Bài 4: Xây dựng lớp Rational chứa các số hữu tỷ gồm các phép toán +, - , *, /, ==, !=, >, >=, <, <=.

Bài 5: Xây dựng lớp Time để lưu trữ giờ, phút, giây gồm các phép toán:

- Phép cộng giữa dữ liệu thời gian và một số nguyên là số giây, kết quả là một dữ liệu thời gian.
- Phép trừ giữa hai dữ liệu thời gian, kết quả là một số nguyên chính là số giây.
- ++ và – để tăng hay giảm thời gian xuống một giây.
- Các phép so sánh.

Bài 6: Xây dựng lớp Date để lưu trữ ngày, tháng, năm gồm các phép toán:

- Phép cộng giữa dữ liệu Date và một số nguyên là số ngày, kết quả là một dữ liệu Date.
- Phép trừ giữa hai dữ liệu Date, kết quả là một số nguyên chính là số ngày.
- ++ và – để tăng hay giảm thời gian xuống một ngày.
- Các phép so sánh.

Bài 7: Các số nguyên 32 bit có thể biểu diễn trong phạm vi từ 2147483648 đến 2147483647. Hãy xây dựng lớp HugeInt để biểu diễn các số nguyên 32 bit gồm các phép toán +, -, *, /

CHƯƠNG 5

TÍNH KẾ THỪA

I. DẪN NHẬP

Trong chương này và chương kế, chúng ta tìm hiểu hai khả năng mà lập trình hướng đối tượng cung cấp là tính kế thừa (inheritance) và tính đa hình (polymorphism). Tính kế thừa là một hình thức của việc sử dụng lại phần mềm trong đó các lớp mới được tạo từ các lớp đã có bằng cách "hút" các thuộc tính và hành vi của chúng và tô điểm thêm với các khả năng mà các lớp mới đòi hỏi. Việc sử dụng lại phần mềm tiết kiệm thời gian trong việc phát triển chương trình. Nó khuyến khích sử dụng lại phần mềm chất lượng cao đã thử thách và gỡ lỗi, vì thế giảm thiểu các vấn đề sau khi một hệ trở thành chính thức. Tính đa hình cho phép chúng ta viết các chương trình trong một kiểu cách chung để xử lý các lớp có liên hệ nhau. Tính kế thừa và tính đa hình các kỹ thuật có hiệu lực đối với sự chia với sự phức tạp của phần mềm.

Khi tạo một lớp mới, thay vì viết các thành viên dữ liệu và các hàm thành viên, lập trình viên có thể thiết kế mà lớp mới được kế thừa các thành viên dữ liệu và các hàm thành viên của lớp trước định nghĩa là lớp cơ sở (base class). Lớp mới được tham chiếu là lớp dẫn xuất (derived class). Mỗi lớp dẫn xuất tự nó trở thành một ứng cử là một lớp cơ sở cho lớp dẫn xuất tương lai nào đó.

Bình thường một lớp dẫn xuất thêm các thành viên dữ liệu và các hàm thành viên, vì thế một lớp dẫn xuất thông thường rộng hơn lớp cơ sở của nó. Một lớp dẫn xuất được chỉ định hơn một lớp cơ sở và biểu diễn một nhóm của các đối tượng nhỏ hơn. Với đối tượng đơn, lớp dẫn xuất, lớp dẫn xuất bắt đầu bên ngoài thực chất giống như lớp cơ sở. Sức mạnh thực sự của sự kế thừa là khả năng định nghĩa trong lớp dẫn xuất các phần thêm, thay thế hoặc tinh lọc các đặc tính kế thừa từ lớp cơ sở.

Mỗi đối tượng của một lớp dẫn xuất cũng là một đối tượng của lớp cơ sở của lớp dẫn xuất đó. Tuy nhiên điều ngược lại không đúng, các đối tượng lớp cơ sở không là các đối tượng của các lớp dẫn xuất của lớp cơ sở đó. Chúng ta sẽ lấy mối quan hệ "đối tượng lớp dẫn xuất là một đối tượng lớp cơ sở" để thực hiện các thao tác quan trọng nào đó. Chẳng hạn, chúng ta có thể luồn một sự đa dạng của các đối tượng khác nhau có liên quan thông qua sư kế thừa thành danh sách liên kết của các đối tượng lớp cơ sở. Điều này cho phép sự đa dạng của các đối tượng để xử lý một cách tổng quát.

Chúng ta phân biệt giữa "là một" (is a) quan hệ và "có một" (has a) quan hệ. "là một" là sự kế thừa. Trong một "là một" quan hệ, một đối tượng của kiểu lớp dẫn xuất cũng có thể được xử lý như một đối tượng của kiểu lớp cơ sở. "có một" là sự phức hợp (composition). Trong một "có một" quan hệ, một đối tượng lớp có một hay nhiều đối tượng của các lớp khác như là các thành viên, do đó lớp bao các đối tượng này gọi là lớp phức hợp (composed class).

II. KẾ THỪA ĐƠN

II.1. Các lớp cơ sở và các lớp dẫn xuất

Thường một đối tượng của một lớp thật sự là một đối tượng của lớp khác cũng được. Một hình chữ nhật là một tứ giác, vì thế lớp *Rectangle* có thể kế thừa từ lớp *Quadrilateral*. Trong khung cảnh này, lớp *Quadrilateral* gọi là một lớp cơ sở và lớp *Rectangle* gọi là một lớp dẫn xuất. Hình 5.1 cho chúng ta một vài ví dụ về kế thừa đơn.

Các ngôn ngữ lập trình hướng đối tượng như SMALLTALK sử dụng thuật ngữ khác: Trong kế thừa, lớp cơ sở được gọi là lớp cha (superclass), lớp dẫn xuất được gọi là lớp con (subclass).

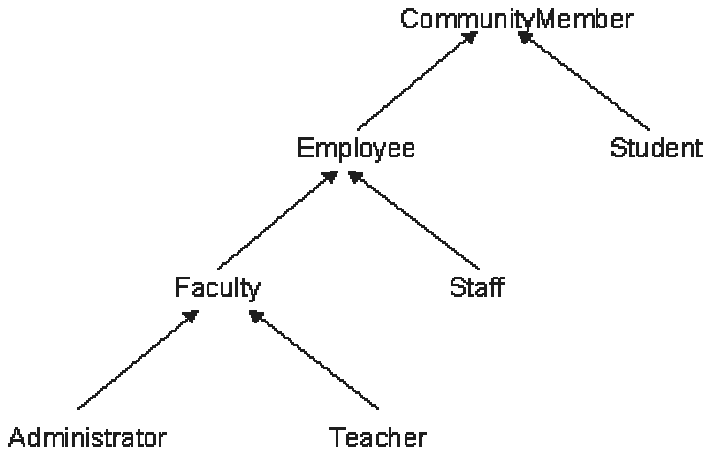
Lớp cơ sở	Lớp dẫn xuất
Student	GraduateStudent
	UndergraduateStudent
Shape	Circle
	Triangle
	Rectangle
Loan	CarLoan

	HomeImprovementLoan
	MortgageLoan
Employee	FacultyMember
	StaffMember
Acount	CheckingAcount
	SavingsAcount

Hình 5.1: Một vài kế thừa đơn.

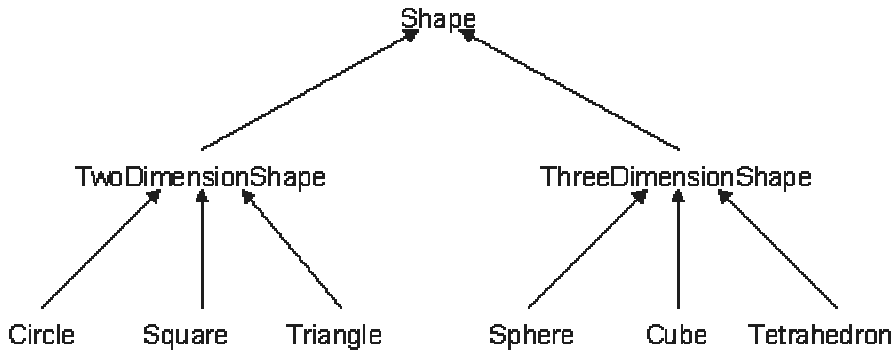
Sự kế thừa hình thành các cấu trúc phân cấp giống cây (còn gọi là cây phả hệ). Một lớp cơ sở tồn tại trong một phân cấp quan hệ với lớp dẫn xuất của nó. Một lớp có thể tồn tại chắc chắn bởi chính nó, nhưng khi một lớp được sử dụng với cơ chế của sự kế thừa thì lớp trở thành hoặc là một lớp cơ sở mà cung cấp các thuộc tính và các hành vi cho các lớp khác, hoặc là lớp trở thành một lớp dẫn xuất mà kế thừa các thuộc tính và các hành vi.

Chúng ta phát triển một phân cấp kế thừa đơn. Một trường đại học cộng đồng đặc thù có hàng ngàn người mà là các thành viên cộng đồng. Những người này gồm các người làm công và các sinh viên. Những người làm công hoặc là các thành viên khoa hoặc các thành viên nhân viên. Các thành viên khoa hoặc là các nhà quản lý hoặc giảng viên. Điều này trở thành phân cấp kế thừa như hình 5.2



Hình 5.2: Một phân cấp kế thừa cho các thành viên của trường đại học cộng đồng.

Phân cấp kế thừa quan trọng khác là phân cấp *Shape* ở hình 5.3.



Hình 5.3: Phân cấp lớp Shape

Để chỉ định lớp *CommissionWorker* được dẫn xuất từ lớp *Employee*, lớp *CommissionWorker* được định nghĩa như sau:

```
class CommissionWorker: public Employee
{…….
};
```

Điều này được gọi là kế thừa **public** và là loại mà phần lớn được sử dụng. Ngoài ra chúng ta còn có kế thừa **private** và kế thừa **protected**. Với kế thừa **public**, các thành viên **public** và **protected** của lớp cơ sở được kế thừa như là các thành viên **public** và **protected** của lớp dẫn xuất tương ứng. Nên nhớ rằng các thành viên **private** của lớp cơ sở không thể truy cập từ các lớp dẫn xuất của lớp đó.

Xử lý các đối tượng lớp cơ sở và các đối tượng lớp dẫn xuất tương tự; phổ biến là được biểu diễn bằng các thuộc tính và các hành vi của lớp cơ sở. Các đối tượng của bất kỳ lớp nào dẫn xuất từ một lớp cơ sở chung có thể tất cả được xử lý như các đối tượng của lớp cơ sở đó.

II.2. Các thành viên protected

Các thành viên **public** của một lớp cơ sở được truy cập bởi tất cả các hàm trong chương trình. Các thành viên **private** của một lớp cơ sở chỉ được truy cập bởi các hàm thành viên và các hàm **friend** của lớp cơ sở.

Truy cập **protected** phục vụ như một mức trung gian của sự bảo vệ giữa truy cập **public** và truy cập **private**. Các thành viên **protected** của một lớp cơ sở có thể chỉ được truy cập bởi các hàm thành viên và các hàm **friend** của lớp cơ sở và bởi các hàm thành viên và các hàm **friend** của lớp dẫn xuất.

Các thành viên lớp dẫn xuất kế thừa **public** có thể tham khảo tới các thành viên **public** và **protected** bằng cách sử dụng các tên thành viên.

II.3. Ép kiểu các con trỏ lớp cơ sở tới các con trỏ lớp dẫn xuất

Một đối tượng của một lớp dẫn xuất kế thừa **public** cũng có thể được xử lý như một đối tượng của lớp cơ sở của nó tương ứng. Nhưng ngược lại không đúng: một đối tượng lớp cơ sở cũng không tự động là một đối tượng lớp dẫn xuất.

Tuy nhiên, có thể sử dụng ép kiểu để chuyển đổi một con trỏ lớp cơ sở thành một con trỏ lớp dẫn xuất.

Ví dụ 5.1: Chương trình sau sẽ được chia thành nhiều file (gồm các file .H và .CPP) và tạo một project có tên là CT5_1.PRJ gồm các file .cpp

File POINT.H:

```
1: //POINT.H
2: //Định nghĩa lớp Point
3: #ifndef POINT_H
4: #define POINT_H
5:
6: class Point
7: {
8: protected:
9: float X,Y;
10: public:
11: Point(float A= 0, float B= 0);
12: void SetPoint(float A, float B);
13: float GetX() const
14: {
15: return X;
16: }
17: float GetY() const
18: {
19: return Y;
20: }
21: friend ostream & operator <<(ostream &Output, const Point &P);
22: };
23:
24: #endif
```

File POINT.CPP

```
1: //POINT.CPP
2: //Định nghĩa các hàm thành viên của lớp Point
3: #include <iostream.h>
4: #include "point.h"
```

```
5:
6: Point::Point(float A, float B)
7: {
8: SetPoint(A, B);
9: }
10:
11: void Point::SetPoint(float A, float B)
12: {
13: X = A;
14: Y = B;
15: }
16:
17: ostream & operator <<(ostream &Output, const Point &P)
18: {
19: Output << '[' << P.X << ", " << P.Y << ']';
20: return Output;
21: }
```

File CIRCLE.H

```
1: //CIRCLE.H
2: //Định nghĩa lớp Circle
3: #ifndef CIRCLE_H
4: #define CIRCLE_H
5:
6: #include "point.h"
7: class Circle : public Point
8: {
9: protected:
10: float Radius;
11: public:
12: Circle(float R = 0.0, float A = 0, float B = 0);
13: void SetRadius(float R);
14: float GetRadius() const;
15: float Area() const;
16: friend ostream & operator <<(ostream &Output, const Circle &C);
17: };
18:
19: #endif
```

File CIRCLE.CPP

```
1: //CIRCLE.CPP
2: //Định nghĩa các hàm thành viên của lớp Circle
3: #include <iostream.h>
4: #include <iomanip.h>
5: #include "circle.h"
6:
7: Circle::Circle(float R, float A, float B): Point(A, B)
8: {
9: Radius = R;
10: }
11:
12: void Circle::SetRadius(float R)
13: {
14: Radius = R;
15: }
16:
17: float Circle::GetRadius() const
18: {
```

```
19: return Radius;
20: }
21:
22: float Circle::Area() const
23: {
24: return 3.14159 * Radius * Radius;
25: }
26:
27: //Xuất một Circle theo dạng: Center = [x, y]; Radius = #.##
28: ostream & operator <<(ostream &Output, const Circle &C)
29: {
30: Output << "Center = [" << C.X << ", " << C.Y
31: << "]; Radius = " << setiosflags(ios::showpoint)
32: << setprecision(2) << C.Radius;
33: return Output;
34: }
```

File CT5_1.CPP:

```
1: //CT5_1.CPP
2: //Chương trình 5.1: Ép các con trỏ lớp cơ sở tới các con trỏ lớp
dẫn xuất
3: #include <iostream.h>
4: #include <iomanip.h>
5: #include "point.h"
6: #include "circle.h"
7:
8: int main()
9: {
10: Point *PointPtr, P(3.5, 5.3);
11: Circle *CirclePtr, C(2.7, 1.2, 8.9);
12: cout << "Point P: "<<P<<endl<<"Circle C: "<<C<< endl;
13 //Xử lý một Circle như một Point (chỉ xem một phần lớp cơ sở)
14: PointPtr = &C;
15: cout << endl << "Circle C (via *PointPtr): "<<*PointPtr<<endl;
16 //Xử lý một Circle như một Circle
17: PointPtr = &C;
18: CirclePtr = (Circle *) PointPtr;
19: cout << endl << "Circle C (via *CirclePtr): " << endl
20:            <<*CirclePtr<< endl << "Area of C (via CirclePtr): "
21:            << CirclePtr->Area() << endl;
22: //Nguy hiểm: Xem một Point như một Circle
23: PointPtr = &P;
24: CirclePtr = (Circle *) PointPtr;
25: cout << endl << "Point P (via *CirclePtr): "<< endl
26:            <<*CirclePtr<< endl << "Area of object CirclePtr
points to: "
27:            <<CirclePtr->Area() << endl;
28: return 0;
29: }
```

Chúng ta chạy ví dụ 5.1, kết quả ở hình 5.4

```
Point P: [3.5, 5.3]
Circle C: Center = [1.2, 8.9]; Radius = 2.70

Circle C (via *PointPtr): [1.20, 8.90]

Circle C (via *CirclePtr):
Center = [1.20, 8.90]; Radius = 2.70
Area of C (via CirclePtr): 22.90

Point P (via *CirclePtr):
Center = [3.50, 5.30]; Radius = 4.02e-38
Area of object CirclePtr points to: 0.00
```

Hình 5.4: Kết quả của ví dụ 5.1

Trong định nghĩa lớp *Point*, các thành viên dữ liệu *X* và *Y* được chỉ định là **protected**, điều này cho phép các lớp dẫn xuất từ lớp *Point* truy cập trực tiếp các thành viên dữ liệu kế thừa. Nếu các thành viên dữ liệu này được chỉ định là **private**, các hàm thành viên **public** của *Point* phải được sử dụng để truy cập dữ liệu, ngay cả bởi các lớp dẫn xuất.

Lớp *Circle* được kế thừa từ lớp *Point* với kế thừa **public** (ở dòng 7 file CIRCLE.H), tất cả các thành viên của lớp *Point* được kế thừa thành lớp *Circle*. Điều này có nghĩa là giao diện **public** bao gồm các hàm thành viên **public** của *Point* cũng như các hàm thành viên *Area()*, *SetRadius()* và *GetRadius()*.

Constructor lớp *Circle* phải bao gồm constructor lớp *Point* để khởi động phần lớp cơ sở của đối tượng lớp *Circle* ở dòng 7 file CIRCLE.CPP, dòng này có thể được viết lại như sau:

```
Circle::Circle(float R, float A, float B)
    : Point(A, B) //Gọi constructor của lớp cơ sở
```

Các giá trị *A* và *B* được chuyển từ constructor lớp *Circle* tới constructor lớp *Point* để khởi động các thành viên *X* và *Y* của lớp cơ sở. Nếu constructor lớp *Circle* không bao gồm constructor lớp *Point* thì constructor lớp *Point* gọi với các giá trị mặc định cho *X* và *Y* (nghĩa là 0 và 0). Nếu lớp *Point* không cung cấp một constructor mặc định thì trình biên dịch phát sinh lỗi.

Trong chương trình chính (file CT5_1.CPP) gán một con trỏ lớp dẫn xuất (địa chỉ của đối tượng *C*) cho con trỏ lớp cơ sở *PointPtr* và xuất đối tượng *C* của *Circle* bằng toán tử chèn dòng của lớp *Point* (ở dòng 14 và 15). Chú ý rằng chỉ phần *Point* của đối tượng *C* của *Circle* được hiển thị. Nó luôn luôn đúng để gán một con trỏ lớp dẫn xuất cho con trỏ lớp cơ sở bởi vì một đối tượng lớp dẫn xuất là một đối tượng lớp cơ sở. Con trỏ lớp cơ sở chỉ trông thấy phần lớp cơ sở của đối tượng lớp dẫn xuất. Trình biên dịch thực hiện một chuyển đổi ngầm của con trỏ lớp dẫn xuất cho một con trỏ lớp cơ sở.

Sau đó chương trình gán một con trỏ lớp dẫn xuất (địa chỉ của đối tượng *C*) cho con trỏ lớp cơ sở *PointPtr* và ép *PointPtr* trở về kiểu *Circle* *. Kết quả của ép kiểu được gán cho *CirclePtr*. Đối tượng *C* của *Circle* được xuất bằng cách sử dụng toán tử chèn dòng của *Circle*. Diện tích của đối tượng *C* được xuất thông qua *CirclePtr*. Các kết quả này là giá trị diện tích đúng bởi vì các con trỏ luôn luôn được trỏ tới một đối tượng *Circle* (từ dòng 17 đến 22).

Kế tiếp, chương trình gán một con trỏ lớp cơ sở (địa chỉ của đối tượng *P*) cho con trỏ lớp cơ sở *PointPtr* và ép *PointPtr* trở về kiểu *Circle* *. Kết quả của ép kiểu được gán cho *CirclePtr*. Đối tượng P được xuất sử dụng toán tử chèn dòng của lớp *Circle*. Chú ý rằng giá trị xuất của thành viên *Radius* "kỳ lạ". Việc xuất một *Point* như một *Circle* đưa đến một giá trị không hợp lệ cho *Radius* bởi vì các con trỏ luôn được trỏ đến một đối tượng *Point*. Một đối tượng *Point* không có một thành viên *Radius*. Vì thế, chương trình xuất giá trị "rác" đối với thành viên dữ liệu *Radius*. Chú ý rằng giá trị của diện tích là 0.0 bởi vì tính toàn này dựa trên giá trị không tồn tại của *Radius* (từ dòng 23 đến 27).Rõ ràng, truy cập các thành viên dữ liệu mà không phải ở đó thì nguy hiểm. Gọi các hàm thành viên mà không tồn tại có thể phá hủy chương trình.

II.4. Định nghĩa lại các thành viên lớp cơ sở trong một lớp dẫn xuất

Một lớp dẫn xuất có thể định nghĩa lại một hàm thành viên lớp cơ sở. Điều này được gọi là overriding. Khi hàm đó được đề cập bởi tên trong lớp dẫn xuất, phiên bản của lớp dẫn xuất được chọn một cách tự động. Toán tử định phạm vi có thể sử dụng để truy cập phiên bản của lớp cơ sở từ lớp dẫn xuất.

II.5. Các lớp cơ sở public, protected và private

Khi dẫn xuất một lớp từ một lớp cơ sở, lớp cơ sở có thể được kế thừa là **public**, **protected** và **private**.

class *<drived_class_name>* **:** *<type_of_inheritance>* *<base_class_name>*
{……..
};

Trong đó *type_of_inheritance* là **public**, **protected** hoặc **private**. Mặc định là **private**.

Khi dẫn xuất một lớp từ một lớp cơ sở **public**, các thành viên **public** của lớp cơ sở trở thành các thành viên **public** của lớp dẫn xuất, và các thành viên **protected** của lớp cơ sở trở thành các thành viên **protected** của lớp dẫn xuất. Các thành viên **private** của lớp cơ sở không bao giờ được truy cập trực tiếp từ một lớp dẫn xuất.

Khi dẫn xuất một lớp từ một lớp cơ sở **protected**, các thành viên **public** và **protected** của lớp cơ sở trở thành các thành viên **protected** của lớp dẫn xuất. Khi dẫn xuất một lớp từ một lớp cơ sở **private**, các thành viên **public** và **protected** của lớp cơ sở trở thành các thành viên **private** của lớp dẫn xuất.

Bảng sau (hình 5.6)tổng kết khả năng truy cập các thành viên lớp cơ sở trong một lớp dẫn xuất dựa trên thuộc tính xác định truy cập thành viên của các thành viên trong lớp cơ sở và kiểu kế thừa.

	Kiểu kế thừa		
Kế thừa **public**	Kế thừa **protected**	Kế thừa **private**	
public	**public** trong lớp dẫn xuất. Có thể truy cập trực tiếp bởi các hàm thành viên không tĩnh, các hàm **friend** và các hàm không thành viên.	**protected** trong lớp dẫn xuất. Có thể truy cập trực tiếp bởi các hàm thành viên không tĩnh, các hàm **friend**.	**private** trong lớp dẫn xuất. Có thể truy cập trực tiếp bởi các hàm thành viên không tĩnh, các hàm **friend**.
protected	**protected** trong lớp dẫn xuất. Có thể truy cập trực tiếp bởi các hàm thành viên không tĩnh, các hàm **friend**.	**protected** trong lớp dẫn xuất. Có thể truy cập trực tiếp bởi các hàm thành viên không tĩnh, các hàm **friend**.	**private** trong lớp dẫn xuất. Có thể truy cập trực tiếp bởi các hàm thành viên không tĩnh, các hàm **friend**.
private	Dấu trong lớp dẫn xuất. Có thể truy cập trực tiếp bởi các hàm thành viên không tĩnh, các hàm **friend** thông qua các hàm thành viên **public** và **protected** của lớp cơ sở.	Dấu trong lớp dẫn xuất. Có thể truy cập trực tiếp bởi các hàm thành viên không tĩnh, các hàm **friend** thông qua các hàm thành viên **public** và **protected** của lớp cơ sở.	Dấu trong lớp dẫn xuất. Có thể truy cập trực tiếp bởi các hàm thành viên không tĩnh, các hàm **friend** thông qua các hàm thành viên **public** và **protected** của lớp cơ sở.

Hình 5.7: Tổng kết khả năng truy cập thành viên lớp cơ sở trong lớp dẫn xuất.

II.6. Các contructor và destructor lớp dẫn xuất

Bởi vì một lớp dẫn xuất kết thừa các thành viên lớp cơ sở của nó (ngoại trừ constructor và destructor), khi một đối tượng của lớp dẫn xuất được khởi động, constructor lớp cơ sở phải được gọi để khởi động các

thành viên lớp cơ sở của đối tượng lớp dẫn xuất. Một bộ khởi tạo lớp cơ sở (sử dụng cú pháp giống như bộ khởi tạo thành viên) có thể được cung cấp trong constructor lớp dẫn xuất để gọi tường minh constructor lớp cơ sở, mặt khác constructor lớp dẫn xuất sẽ gọi constructor mặc định lớp cơ sở.

Các constructor lớp cơ sở và các toán tử gán lớp cơ sở không được kế thừa bởi lớp dẫn xuất.Tuy nhiên, các constructor và các toán tử gán lớp dẫn xuất có thể gọi các constructor và các toán tử gán lớp cơ sở.

Một constructor lớp dẫn xuất luôn gọi constructor lớp cơ sở của nó đầu tiên để khởi tạo các thành viên lớp cơ sở của lớp dẫn xuất. Nếu constructor lớp dẫn bị bỏ qua, constructor mặc định lớp dẫn gọi constructor lớp cơ sở. Các destructor được gọi theo thứ tự ngược lại thứ tự gọi các constructor, vì thế destructor lớp dẫn xuất được gọi trước destructor lớp cơ sở của nó.

Ví dụ 5.4: Minh họa thứ tự các contructor và destructor lớp cơ sở và lớp dẫn xuất được gọi và project có tên là CT5_4.PRJ

File POINT.H

```
1: //POINT.H
2: //Định nghĩa lớp Point
3: #ifndef POINT_H
4: #define POINT_H
5:
6: class Point
7: {
8: public:
9: Point(float A= 0.0, float B= 0.0);
10: ~Point();
11: protected:
12: float X, Y;
13: };
14:
15: #endif
```

File POINT.CPP

```
1: //POINT.CPP
2: //Định nghĩa các hàm thành viên lớp Point
3: #include <iostream.h>
4: #include "point.h"
5:
6: Point::Point(float A, float B)
7: {
8: X = A;
9: Y = B;
10: cout << "Point constructor: "
11:            << '[' << X << ", " << Y << ']' << endl;
12: }
13:
14: Point::~Point()
15: {
16: cout << "Point destructor: "
17:            << '[' << X << ", " << Y << ']' << endl;
18: }
```

File CIRCLE.H

```
1: //CIRCLE.H
```

```
2: //Định nghĩa lớp Circle
3: #ifndef CIRCLE_H
4: #define CIRCLE_H
5:
6: #include "point.h"
7: #include <iomanip.h>
8:
9: class Circle : public Point
10: {
11: public:
12: Circle(float R = 0.0, float A = 0, float B = 0);
13: ~Circle();
14: private:
15: float Radius;
16: };
17:
18: #endif
```

File CIRCLE.CPP

```
1: //CIRCLE.CPP
2: //Định nghĩa các hàm thành viên lớp Circle
3: #include "circle.h"
4:
5: Circle::Circle(float R, float A, float B): Point(A, B)
6: {
7: Radius = R;
8: cout << "Circle constructor: Radius is "
9:            << Radius << " [" << A << ", " << B << ']' << endl;
10: }
11:
12: Circle::~Circle()
13: {
14: cout << "Circle destructor: Radius is "
15:             << Radius << " [" << X << ", " << Y << ']' << endl;
16: }
```

File CT5_4.CPP

```
1: //CT5_4.CPP
2: //Chương trình 5.4
3: #include <iostream.h>
4: #include "point.h"
5: #include "circle.h"
6: int main()
7: {
8: {
9: Point P(1.1, 2.2);
10: }
11: cout << endl;
12: Circle C1(4.5, 7.2, 2.9);
13: cout << endl;
14: Circle C2(10, 5, 5);
15: cout << endl;
16: return 0;
17: }
```

Chúng ta chạy ví dụ 5.4, kết quả ở hình 5.8

```
Point constructor: [1.1, 2.2]
Point destructor:  [1.1, 2.2]

Point constructor: [7.2, 2.9]
Circle constructor: Radius is 4.5 [7.2, 2.9]

Point constructor: [5, 5]
Circle constructor: Radius is 10 [5, 5]

Circle destructor:  Radius is 10 [5, 5]
Point destructor:  [5, 5]
Circle destructor:  Radius is 4.5 [7.2, 2.9]
Point destructor:  [7.2, 2.9]
```

Hình 5.8: Kết quả của ví dụ 5.4

II.7. Chuyển đổi ngầm định đối tượng lớp dẫn xuất sang đối tượng lớp cơ sở

Mặc dù một đối tượng lớp dẫn xuất cũng là một đối tượng lớp cơ sở, kiểu lớp dẫn xuất và kiểu lớp cơ sở thì khác nhau. Các đối tượng lớp dẫn xuất có thể được xử lý như các đối tượng lớp cơ sở. Điều này có ý nghĩa bởi vì lớp dẫn xuất có các thành viên tương ứng với mỗi thành viên của lớp cơ sở. Phép gán theo chiều hướng ngược lại là không cho phép bởi vì gán một đối tượng lớp cơ sở cho đối tượng lớp dẫn xuất sẽ cho phép thêm các thành viên lớp dẫn xuất không xác định.

Một con trỏ trỏ tới một đối tượng lớp dẫn xuất có thể được chuyển đổi ngầm định thành một con trỏ trỏ tới một đối tượng lớp cơ sở bởi vì một đối tượng lớp dẫn xuất là một đối tượng lớp cơ sở.

Có bốn cách để trộn và đối sánh các con trỏ lớp cơ sở và các con trỏ lớp dẫn xuất với các đối tượng lớp cơ sở và các đối tượng lớp dẫn xuất:

- Tham chiếu tới một đối tượng lớp cơ sở với một con trỏ lớp cơ sở thì không phức tạp.
- Tham chiếu tới một đối tượng lớp dẫn xuất với một con trỏ lớp dẫn xuất thì không phức tạp.
- Tham chiếu tới đối tượng lớp dẫn xuất với một con trỏ lớp cơ sở thì an toàn bởi vì đối tượng lớp dẫn xuất cũng là một đối tượng lớp cơ sở của nó. Như vậy đoạn mã chỉ có thể tham chiếu tới các thành viên lớp cơ sở. Nếu đoạn mã tham chiếu tới các thành viên lớp dẫn xuất thông qua con trỏ lớp cơ sở, trình biên dịch sẽ báo một lỗi về cú pháp.
- Tham chiếu tới một đối tượng lớp cơ sở với một con trỏ lớp dẫn xuất thì có lỗi cú pháp. Đầu tiên con trỏ lớp dẫn xuất phải được ép sang con trỏ lớp cơ sở.

III. ĐA KẾ THỪA (MULTIPLE INHERITANCE)

Một lớp có thể được dẫn xuất từ nhiều lớp cơ sở, sự dẫn xuất như vậy được gọi là đa kế thừa. Đa kế thừa có nghĩa là một lớp dẫn xuất kế thừa các thành viên của các lớp cơ sở khác nhau. Khả năng mạnh này khuyến khích các dạng quan trọng của việc sử dụng lại phần mềm, nhưng có thể sinh ra các vấn đề nhập nhằng.

Ví dụ 5.7: Lớp Circle và project có tên là CT5_8.PRJ (gồm các file DIRIVED.CPP, CT5_8.CPP).

File BASE1.H

```
1: //BASE1.H
2: //Định nghĩa lớp Base1
3: #ifndef BASE1_H
4: #define BASE1_H
5:
6: class Base1
7: {
8: protected:
```

```
9: int Value;
10: public:
11: Base1(int X)
12: {
13: Value = X;
14: }
15: int GetData() const
16: {
17: return Value;
18: }
19: };
20:
21: #endif
```

File BASE2.H

```
1: //BASE2.H
2: //Định nghĩa lớp Base2
3: #ifndef BASE2_H
4: #define BASE2_H
5:
6: class Base2
7: {
8: protected:
9: char Letter;
10: public:
11: Base2(char C)
12: {
13: Letter = C;
14: }
15: char GetData() const
16: {
17: return Letter;
18: }
19: };
20:
21: #endif
```

File DERIVED.H

```
1: //DERIVED.H
2: //Định nghĩa lớp Derived mà kế thừa từ nhiều lớp cơ sở (Base1 &
Base2)
3: #ifndef DERIVED_H
4: #define DERIVED_H
5:
6: #include "base1.h"
7: #include "base2.h"
8:
9: class Derived : public Base1, public Base2
10: {
11: private:
12: float Real;
13: public:
14: Derived(int, char, float);
15: float GetReal() const;
16: friend ostream & operator <<(ostream &Output, const Derived &D);
17: };
18:
19: #endif
```

File DERIVED.CPP

```
1: //DERIVED.H
2: //Định nghĩa lớp Derived kế thừa từ nhiều lớp cơ sở (Base1 & Base2)
3: #ifndef DERIVED_H
4: #define DERIVED_H
5:
6: #include "base1.h"
7: #include "base2.h"
8:
9: class Derived : public Base1, public Base2
10: {
11: private:
12: float Real;
13: public:
14: Derived(int, char, float);
15: float GetReal() const;
16: friend ostream & operator <<(ostream &Output, const Derived &D);
17: };
18:
19: #endif
```

File CT5_8.CPP

```
1: //CT5_8.CPP
2: //Chương trình 5.8
3: #include <iostream.h>
4: #include "base1.h"
5: #include "base2.h"
6: #include "derived.h"
7:
8: int main()
9: {
10: Base1 B1(10), *Base1Ptr;
11: Base2 B2('Z'), *Base2Ptr;
12: Derived D(7, 'A', 3.5);
13: cout << "Object B1 contains integer "
14:      << B1.GetData() << endl
15:      << "Object B2 contains character "
16:      << B2.GetData() << endl
17:      << "Object D contains:" << endl << D << endl << endl;
18: cout << "Data members of Derived can be"
19:      << " accessed individually:" << endl
20:      << " Integer: " << D.Base1::GetData() << endl
21:      << " Character: " << D.Base2::GetData() << endl
22:      << "Real number: " << D.GetReal() << endl << endl;
23: cout << "Derived can be treated as an "
24:      << "object of either base class:" << endl;
25: Base1Ptr = &D;
26: cout << "Base1Ptr->GetData() yields "
27:      << Base1Ptr->GetData() << endl ;
28: Base2Ptr = &D;
29: cout << "Base2Ptr->GetData() yields "
30:      << Base2Ptr->GetData() << endl;
31: return 0;
32: }
```

Chúng ta chạy ví dụ 5.8, kết quả ở hình 5.13

```
Object B1 contains integer 10
Object B2 contains character Z
Object D contains:
   Integer: 7
 Character: A
Real number: 3.5

Data members of Derived can be accessed individually:
   Integer: 7
 Character: A
Real number: 3.5

Derived can be treated as an object of either base class:
Base1Ptr->GetData() yields 7
Base2Ptr->GetData() yields A
```

Hình 5.13: Kết quả của ví dụ 5.8

Việc kế thừa nhiều lớp cơ sở tạo ra một loạt các điểm nhập nhằng trong các chương trình C++. Chẳng hạn, trong các chương trình của ví dụ 5.8, nếu thực hiện lệnh:

D.GetData();

thì trình biên dịch (Borland C++ 3.1) sẽ báo lỗi:

Member is ambiguous: 'Base1::GetData' and 'Base1::GetData'

Bởi vì lớp *Derived* kế thừa hai hàm khác nhau có cùng tên là *GetData()* từ hai lớp cơ sở của nó. *Base1::GetData()* là hàm thành viên **public** của lớp cơ sở **public**, và nó trở thành một hàm thành viên **public** của *Derived*. *Base2::GetData()* là hàm thành viên **public** của lớp cơ sở **public**, và nó trở thành một hàm thành viên **public** của *Derived*. Do đó trình biên dịch không thể xác định hàm thành viên *GetData()* của lớp cơ sở nào để gọi thực hiện. Vì vậy, chúng ta phải sử dụng tên lớp cơ sở và toán tử định phạm vi để xác định hàm thành viên của lớp cơ sở lúc gọi hàm *GetData()*.

Cú pháp của một lớp kế thừa nhiều lớp cơ sở:

class *<drived_class_name>* **:** *<type_of_inheritance> <base_class_name1>* **,**

<type_of_inheritance> <base_class_name2>, …

{

……..

};

Trình tự thực hiện constructor trong đa kế thừa: constructor lớp cở sở xuất hiện trước sẽ thực hiện trước và cuối cùng mới tới constructor lớp dẫn xuất. Đối với destructor có trình tự thực hiện theo thứ tự ngược lại.

IV. CÁC LỚP CƠ SỞ ẢO (VIRTUAL BASE CLASSES)

Chúng ta không thể khai báo hai lần cùng một lớp trong danh sách của các lớp cơ sở cho một lớp dẫn xuất. Tuy nhiên vẫn có thể có trường hợp cùng một lớp cơ sở được đề cập nhiều hơn một lần trong các lớp tổ tiên của một lớp dẫn xuất. Điều này phát sinh lỗi vì không có cách nào để phân biệt hai lớp cơ sở gốc.

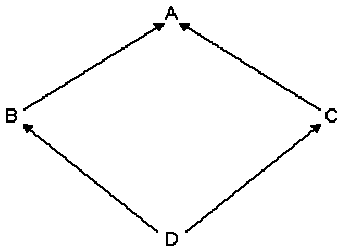
Ví dụ 5.9:

```
1: //Chương trình 5.9
2: #include <iostream.h>
3: class A
4: {
5: public:
6: int X1;
```

```
7: };
8:
9: class B : public A
10: {
11: public:
12: float X2;
13: };
14:
15: class C : public A
16: {
17: public:
18: double X3;
19: };
20:
21: class D : public B,public C
22: {
23: public:
24: char X4;
25: };
26:
27: int main()
28: {
29: D Obj;
30: Obj.X2=3.14159F;
31: Obj.X1=0; //Nhập nhằng
32: Obj.X4='a';
33: Obj.X3=1.5;
34: cout<<"X1="<<Obj.X1<<endl; //Nhập nhằng
35: cout<<"X2="<<Obj.X2<<endl;
36: cout<<"X3="<<Obj.X3<<endl;
37: cout<<"X4="<<Obj.X4<<endl;
38: return 0;
39: }
```

Khi biên dịch chương trình ở ví dụ 5.9, trình biên dịch sẽ báo lỗi ở dòng 31 và 34:

Member is ambiguous: 'A::X1' and 'A::X1'



Hình 5.14

Ở đây chúng ta thấy có hai lớp cở sở *A* cho lớp *D*, và trình biên dịch không thể nào nhận biết được việc truy cập *X1* được kế thừa thông qua *B* hoặc truy cập *X1* được kế thừa thông qua C. Để khắc phục điều này, chúng ta chỉ định một cách tường minh trong lớp *D* như sau:

Obj.C::X1=0;

Tuy nhiên, đây cũng chỉ là giải pháp có tính chấp vá, bởi thực chất *X1* nào trong trường hợp nào cũng được. Giải pháp cho vấn đề này là khai báo *A* như lớp cơ sở kiểu **virtual** cho cả *B* và *C*. Khi đó chương trình ở ví dụ 5.9 được viết lại như sau:

Ví dụ 5.10:

```
#include <iostream.h>
class A
   {
   public:
         int X1;
   };
class B : virtual public A
{
   public:
         float X2;
};
class C : virtual public A
{
   public:
         double X3;
};
class D : public B,public C
{
   public:
         char X4;
};
int main()
{
   D Obj;
   Obj.X2=3.14159F;
   Obj.X1=0; //OK
   Obj.X4='a';
   Obj.X3=1.5;
   cout<<"X1="<<Obj.X1<<endl; //OK
   cout<<"X2="<<Obj.X2<<endl;
   cout<<"X3="<<Obj.X3<<endl;
   cout<<"X4="<<Obj.X4<<endl;
   return 0;
}
```

Chúng ta chạy ví dụ 5.10, kết quả ở hình 5.15

```
X1=0
X2=3.14159
X3=1.5
X4=a
```

Hình 5.15: Kết quả của ví dụ 5.10

Các lớp cơ sở kiểu **virtual**, có cùng một kiểu lớp, sẽ được kết hợp để tạo một lớp cơ sở duy nhất có kiểu đó cho bất kỳ lớp dẫn xuất nào kế thừa chúng. Hai lớp cơ sở A ở trên bất giờ sẽ trở thành một lớp cơ sở A duy nhất cho bất kỳ lớp dẫn xuất nào từ B và C. Điều này có nghĩa là D chỉ có một cơ sở của lớp A, vì vậy tránh được sự nhập nhằng.

BÀI TẬP

Bài 1: Xây dựng lớp Stack với các thao tác cần thiết. Từ đó hãy dẫn xuất từ lớp Stack để đổi một số nguyên dương sang hệ đếm bất kỳ.

Bài 2: Hãy xây dựng các lớp cần thiết trong phân cấp hình 5.2

Bài 3: Hãy xây dựng các lớp cần thiết trong phân cấp hình 5.3 để tính diện tích (hoặc diện tích xung quanh) và thể tích.

Bài 4: Viết một phân cấp kế thừa cho các lớp Quadrilateral (hình tứ giác), Trapezoid (hình thang), Parallelogram (hình bình hành), Rectangle (hình chữ nhật), và Square (hình vuông). Trong đó Quadrilateral là lớp cơ sở của phân cấp.

CHƯƠNG 6

TÍNH ĐA HÌNH

I. DẪN NHẬP

Tính đa hình (polymorphism) là khả năng thiết kế và cài đặt các hệ thống mà có thể mở rộng dễ dàng hơn. Các chương trình có thể được viết để xử lý tổng quát – như các đối tượng lớp cơ sở – các đối tượng của tất cả các lớp tồn tại trong một phân cấp. Khả năng cho phép một chương trình sau khi đã biên dịch có thể có nhiều diễn biến xảy ra là một trong những thể hiện của tính đa hình – tính muôn màu muôn vẻ – của chương trình hướng đối tượng, một thông điệp được gởi đi (gởi đến đối tượng) mà không cần biết đối tượng nhận thuộc lớp nào. Để thực hiện được tính đa hình, các nhà thiết kế C++ cho chúng ta dùng cơ chế kết nối động (dynamic binding) thay cho cơ chế kết nối tĩnh (static binding) ngay khi chương trình biên dịch được dùng trong các ngôn ngữ cổ điển như C, Pascal, …

II. PHƯƠNG THỨC ẢO (VIRTUAL FUNCTION)

Khi xây dựng các lớp của một chương trình hướng đối tượng để tạo nên một cấu trúc phân cấp hoặc cây phả hệ, người lập trình phải chuẩn bị các hành vi giao tiếp chung của các lớp đó. Hành vi giao tiếp chung sẽ được dùng để thể hiện cùng một hành vi, nhưng có các hành động khác nhau, đó chính là phương thức ảo. Đây là một phương thức tồn tại để có hiệu lực nhưng không có thực trong lớp cơ sở, còn trong các lớp dẫn xuất. Như vậy phương thức ảo chỉ được xây dựng khi có một hệ thống cây phả hệ. Phương thức này sẽ được gọi thực hiện từ thực thể của lớp dẫn xuất nhưng mô tả về chúng trong lớp cơ sở.

Chúng ta khai báo phương thức ảo bằng thêm từ khóa **virtual** ở phía trước. Khi đó các phương thức có cùng tên với phương thức này trong các lớp dẫn xuất cũng là phương thức ảo.

Ví dụ 6.1:

```
1: //Chương trình 6.1
2: #include <iostream.h>
3:
4: class Base
5: {
6: public:
7: virtual void Display()
8: {
9: cout<<"class Base"<<endl;
10: }
11: };
12:
13: class Derived : public Base
14: {
15: public:
16: virtual void Display()
17: {
18: cout<<"class Derived"<<endl;
19: }
20: };
21:
21: void Show(Base *B)
22: {
23: B->Display(); //Con trỏ B chỉ đến phương thức Display() nào
(của lớp Base
24 //hoặc lớp Derived) tùy vào lúc chạy chương trình.
25: }
26: int main()
27: {
28: Base *B=new Base;
29: Derived *D=new Derived;
```

```
30: B->Display(); //Base::Display()
31: D->Display(); //Derived::Display()
32: Show(B); //Base::Display()
33: Show(D); //Derived::Display()
34: return 0;
35: }
```

Chúng ta chạy ví dụ 6.1, kết quả ở hình 6.1

```
class Base
class Derived
class Base
class Derived
```

Hình 6.1: Kết quả của ví dụ 6.1

Trong ví dụ 6.1, lớp cơ sở *Base* có phương thức *Display()* được khai báo là phương thức ảo. Phương thức này trong lớp dẫn xuất *Derived* được định nghĩa lại nhưng cũng là một phương thức ảo. Thật ra, không ra không có khai báo **virtual** cho phương thức *Display()* của lớp *Derived* cũng chẳng sao, trình biên dịch vẫn hiểu đó là phương thức ảo. Tuy nhiên, khai báo **virtual** rõ ràng ở các lớp dẫn xuất làm cho chương trình trong sáng, dễ hiểu hơn. Hai dòng 30 và 31, chúng ta biết chắc phương thức *Display()* của lớp nào được gọi (của lớp *Base* hoặc lớp *Derived*). Nhưng hai dòng 32 và 33, nếu không có cơ chế kết nối động, chúng ta đoán rằng việc gọi hàm *Show()* sẽ luôn luôn kéo theo phương thức *Base::Display()*. Quả vậy, bỏ đi khai báo **virtual** cho phương thức *Base::Display()*, khi đó dòng lệnh: **Show(D);**

gọi đến *Base::Display()* vì đối tượng lớp dẫn xuất cũng là đối tượng lớp cơ sở (nghĩa là tdb tự động chuyển đổi kiểu: đối tượng *D* kiểu *Derived* chuyển thành kiểu *Base*.

Nhờ khai báo **virtual** cho phương thức *Base::Display()* nên sẽ không thực hiện gọi phương thức *Base::Display()* một cách cứng nhắc trong hàm *Show()* mà chuẩn bị một cơ chế mềm dẻo cho việc gọi phương thức *Display()* tùy thuộc vào sự xác định kiểu của tham số vào lúc chạy chương trình.

Cơ chế đó ra sao? Khi nhận thấy có khai báo **virtual** trong lớp cơ sở, trình biên dịch sẽ thêm vào mỗi đối tượng của lớp cơ sở và các lớp dẫn xuất của nó một con trỏ chỉ đến bảng phương thức ảo (virtual function table). Con trỏ đó có tên là vptr (virtual pointer). Bảng phương thức ảo là nơi chứa các con trỏ chỉ đến đoạn chương trình đã biên dịch ứng với các phương thức ảo. Mỗi lớp có một bảng phương thức ảo. Trình biên dịch chỉ lập bảng phương thức ảo khi bắt đầu có việc tạo đối tượng của lớp. Đến khi chương trình chạy, phương thức ảo của đối tượng mới được nối kết và thi hành thông qua con trỏ vptr.

Trong ví dụ 6.1, lệnh gọi hàm: **Show(D);**

Đối tượng *D* thuộc lớp *Derived* tuy bị chuyển đổi kiểu thành một đối tượng thuộc lớp *Base* nhưng nó không hoàn toàn giống một đối tượng của *Base* chính cống như *B*. Nếu như con trỏ vptr trong *B* chỉ đến vị trí trên bảng phương thức ảo ứng với phương thức *Base::Display()*, thì con trỏ vptr trong *D* vẫn còn chỉ đến phương thức *Derived::Display()* cho dù *D* bị chuyển kiểu thành *Base*. Đó là lý do tại sao lệnh: **Show(D);**

gọi đến phương thức *Derived::Display()*.

Các đặc trưng của phương thức ảo:

- Phương thức ảo không thể là các hàm thành viên tĩnh.
- Một phương thức ảo có thể được khai báo là **friend** trong một lớp khác nhưng các hàm **friend** của lớp thì không thể là phương thức ảo.
- Không cần thiết phải ghi rõ từ khóa **virtual** khi định nghĩa một phương thức ảo trong lớp dẫn xuất (để cũng chẳng ảnh hưởng gì).
- Để sự kết nối động được thực hiện thích hợp cho từng lớp dọc theo cây phả hệ, một khi phương thức nào đó đã được xác định là ảo, từ lớp cơ sở đến các lớp dẫn xuất đều phải định nghĩa thống nhất về tên, kiểu

trả về và danh sách các tham số. Nếu đối với phương thức ảo ở lớp dẫn xuất, chúng ta lại sơ suất định nghĩa các tham số khác đi một chút thì trình biên dịch sẽ xem đó là phương thức khác. Đây chính là điều kiện để kết nối động.

Ví dụ 6.2:

```
2: #include <iostream.h>
3:
4: class Base
5: {
6: public:
7: virtual void Print(int A,int B);
8: };
9:
10: class Derived : public Base
11: {
12: public:
13: virtual void Print(int A,double D);
14: };
15:
16: void Base::Print(int A,int B)
17: {
18: cout<<"A="<<A<<",B="<<B<<endl;
19: }
20:
21: void Derived::Print(int A,double D)
22: {
23: cout<<"A="<<A<<",D="<<D<<endl;
24: }
25:
26: void Show(Base *B)
27: {
28: B->Print(3,5);
29: }
30:
31: int main()
32: {
33: Base *B=new Base;
34: Derived *D=new Derived;
35: Show(B); //Base::Print()
36: Show(D); //Base::Print()
37: return 0;
38: }
```

Chúng ta chạy ví dụ 6.2, kết quả ở hình 6.2

```
A=3,B=5
A=3,B=5
```

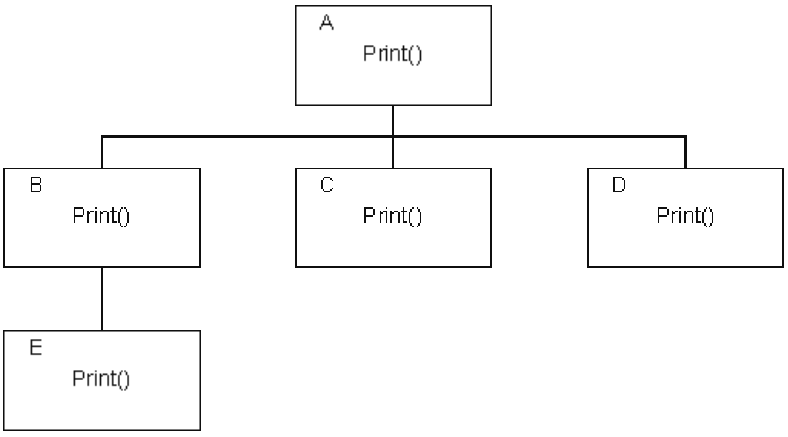
Hình 6.2: Kết quả của ví dụ 6.2

Trong ví dụ 6.2, trong lớp cơ sở *Base* và lớp dẫn xuất *Derived* đều có phương thức ảo *Print()*. Nhưng quan sát kỹ chúng ta, phương thức *Print()* trong lớp *Derived* có tham số thứ hai khác kiểu với phương thức *Print()* trong lớp *Base*. Vì thế, chúng ta không thể chờ đợi lệnh ở dòng 36 sẽ gọi đến phương thức *Derived::Print(int,double)*. Phương thức *Derived::Print(int,double)* nằm ngoài đường dây phương thức ảo nên hàm *Show()* chỉ luôn gọi đến phương thức *Derived::Print(int,int)* mà thôi. Do có khai báo virtual đối với phương thức *Derived::Print(int,double)*, chúng ta có thể nói phương thức này sẽ mở đầu cho một đường dây phương thức ảo *Print(int,double)* mới nếu sau lớp *Derived* còn có các lớp dẫn xuất của nó.

III. LỚP TRỪU TƯỢNG (ABSTRACT CLASS)

Trong quá trình thiết kế chương trình theo hướng đối tượng, để tạo nên một hệ thống phả hệ mang tính kế thừa cao, người lập trình phải đoán trước sự phát triển của cấu trúc, từ đó chọn lựa những thành viên phù hợp cho các lớp ở trên cùng. Rõ ràng đây là một công việc vô cùng khó khăn. Để tránh tình trạng người lập trình xây dựng các đối tượng lãng phí bộ nhớ, ngôn ngữ C++ cho phép chúng ta thiết kế các lớp có các không phương thức ảo không làm gì cả, và cũng không thể tạo ra đối tượng thuộc lớp đó. Những lớp như vậy gọi là lớp trừu tượng.

Trong cấu trúc trên hình 6.3, không phải lớp nào cũng thực sự cần đến phương thức *Print()*, nhưng nó có mặt khắp nơi để tạo ra bộ mặt chung cho mọi lớp trong cấu trúc cây. Phương thức của lớp trên cùng như *A::Print()* thường là phương thức ảo để có được tính đa hình.



Hình 6.3

Nhờ đó, với hàm sau:

```
void Show(A* a)
{
a->Print();
}
```

chúng ta có thể truyền đối tượng đủ kiểu cho nó (*A*, *B*, *C*, *D* hoặc *E*) mà vẫn gọi đến đúng phương thức Print() phù hợp dù kiểu của đối tượng lúc biên dịch vẫn còn chưa biết. Với vai trò "lót đường" như vậy, phương thức *A::Print()* có thể chẳng có nội dung gì cả

```
class A
{
        public:
                virtual void Print()
                {
                }
};
```

Khi đó người ta gọi phương thức *A::Print()* là phương thức ảo rỗng (null virtual function), nó chẳng làm gì hết. Tuy nhiên lớp *A* vẫn là một lớp bình thường, chúng ta có thể tạo ra một đối tượng thuộc nó, có thể truy cập tới phương thức *A::Print()*. Để tránh trình trạng vô tình tạo ra đối tượng thuộc lớp này, người ta thường xây dựng lớp trừu tượng, trình biên dịch cho phép tạo ra lớp có các phương thức thuần ảo (pure virtual function) như sau:

```
class A
{
        public:
                virtual void Print() = 0;
};
```

Phương thức ảo *Print()* bây giờ là phương thức thuần ảo – phương thức có tên được gán bởi giá trị zero. Lớp *A* chứa phương thức thuần ảo được gọi là lớp trừu tượng.

Ví dụ 6.3:

```
1: //Chương trình 6.3
2: #include <iostream.h>
3:
4: class A
5: {
6: public:
7: virtual void Print()=0; //Phương thức thuần ảo
8: };
9:
10: class B : public A
11: {
12: public:
13: virtual void Print()
14: {
15: cout<<"Class B"<<endl;
16: }
17: };
18:
19: class C : public B
20: {
21: public:
22: virtual void Print()
23: {
24: cout<<"Class C"<<endl;
25: }
26: };
27:
28: void Show(A *a)
29: {
30: a->Print();
31: }
32:
33: int main()
34: {
35: B *b=new B;
36: C *c=new C;
37: Show(b); //B::Print()
38: Show(c); //C::Print()
39: return 0;
40: }
```

Chúng ta chạy ví dụ 6.3, kết quả ở hình 6.4

```
Class B
Class C
```

Hình 6.4: Kết quả của ví dụ 6.3

Lớp *A* được tạo ra để làm cơ sở cho việc hình thành các lớp con cháu (*B* và *C*). Nhờ có Phương thức ảo *Print()* từ lớp *A* cho tới lớp *C*, tính đa hình được thể hiện.

Lưu ý:

- Chúng ta không thể tạo ra một đối tượng của lớp trừu tượng, nhưng hoàn toàn có thể tạo ra một con trỏ trỏ đến lớp này (vì con trỏ không phải là đối tượng thuộc lớp) hoặc là một tham chiếu.

- Nếu trong lớp kế thừa từ lớp trừu tượng chúng ta không định nghĩa phương thức thuần ảo, do tính kế thừa nó sẽ bao hàm phương thức thuần ảo của lớp cơ sở, nên lớp dẫn xuất này sẽ trở thành lớp trừu tượng.

- Theo định nghĩa lớp trừu tượng, nếu trong lớp dẫn xuất (từ lớp cơ sở trừu tượng) chúng ta định nghĩa thêm một phương thức thuần ảo khác, lớp này cũng sẽ trở thành lớp trừu tượng.

IV. CÁC THÀNH VIÊN ẢO CỦA MỘT LỚP

IV.1. Toán tử ảo

Toán tử thực chất cũng là một hàm nên chúng ta có thể tạo ra các toán tử ảo trong một lớp. Tuy nhiên do đa năng hóa khi tạo một toán tử cần chú ý đến các kiểu của các toán hạng phải sử dụng kiểu của lớp cơ sở gốc có toán tử ảo.

Ví dụ 6.4: Đa năng hóa toán tử với hàm toán tử là phương thức ảo.

```
1: //Chương trình 6.4: Toán tử ảo
2: #include <iostream.h>
3:
4: class A
5: {
6: protected:
7: int X1;
8: public:
9: A(int I)
10: {
11: X1=I;
12: }
13: virtual A& operator + (A& T);
14: virtual A& operator = (A& T);
15: virtual int GetA()
16: {
17: return X1;
18: }
19: virtual int GetB()
20: {
21: return 0;
22: }
23: virtual int GetC()
24: {
25: return 0;
26: }
27: void Print(char *St)
28: {
29: cout<<St<<":X1="<<X1<<endl;
30: }
31: };
32:
33: class B : public A
34: {
35: protected:
36: int X2;
37: public:
38: B(int I,int J):A(I)
39: {
40: X2=J;
41: }
42: virtual A& operator + (A& T);
43: virtual A& operator = (A& T);
44: virtual int GetB()
```

```
45: {
46: return X2;
47: }
48: void Print(char *St)
49: {
50: cout<<St<<":X1="<<X1<<",X2="<<X2<<endl;
51: }
52: };
53:
54: class C : public B
55: {
56: protected:
57: int X3;
58: public:
59: C(int I,int J,int K):B(I,J)
60: {
61: X3=K;
62: }
63: virtual A& operator + (A& T);
64: virtual A& operator = (A& T);
65: virtual int GetC()
66: {
67: return X3;
68: }
69: void Print(char *St)
70: {
71: cout<<St<<":X1="<<X1<<",X2="<<X2<<",X3="<<X3<<endl;
72: }
73: };
74:
75: A& A::operator + (A& T)
76: {
77: X1+=T.GetA();
78: return *this;
79: }
80:
81: A& A::operator = (A& T)
82: {
83: X1=T.GetA();
84: return *this;
85: }
86:
87: A& B::operator + (A& T)
88: {
89: X1+=T.GetA();
90: X2+=T.GetB();
91: return *this;
92: }
93:
94: A& B::operator = (A& T)
95: {
96: X1=T.GetA();
97: X2=T.GetB();
98: return *this;
99: }
100:
101:A& C::operator + (A& T)
102: {
```

```
103: X1+=T.GetA();
104: X2+=T.GetB();
105: X3+=T.GetC();
106: return *this;
107 }
108:
109: A& C::operator = (A& T)
110: {
111: X1=T.GetA();
112: X2=T.GetB();
113: X3=T.GetC();
114: return *this;
115: }
116:
117: void AddObject(A& T1,A& T2)
118: {
119: T1=T1+T2;
120: }
121:
122: int main()
123: {
124: A a(10);
125: B b(10,20);
126: C c(10,20,30);
127: a.Print("a");
128: b.Print("b");
129: c.Print("c");
130: AddObject(a,b);
131: a.Print("a");
132: AddObject(b,c);
133: b.Print("b");
134: AddObject(c,a);
135: c.Print("c");
136: a=b+c;
137: a.Print("a");
138: c=c+a;
139: c.Print("c");
140: return 0;
141: }
```

Chúng ta chạy ví dụ 6.4, kết quả ở hình 6.5

```
a:X1=10
b:X1=10,X2=20
c:X1=10,X2=20,X3=30
a:X1=20
b:X1=20,X2=40
c:X1=30,X2=20,X3=30
a:X1=50
c:X1=80,X2=20,X3=30
```

Hình 6.5: Kết quả của ví dụ 6.4

IV.2. Có constructor và destructor ảo hay không?

Khi một đối tượng thuộc lớp có phương thức ảo, để thực hiện cơ chế kết nối động, trình biên dịch sẽ tạo thêm một con trỏ vptr như một thành viên của lớp, con trỏ này có nhiệm vụ quản lý địa chỉ của phương thức ảo. Một lớp chỉ có một bảng phương thức ảo, trong khi đó có thể có nhiều đối tượng thuộc lớp, nên khi một

đối tượng khác thuộc cùng lớp tạo ra thì con trỏ vptr đã còn tại. Chính vì vậy bảng phương thức ảo phải được tạo ra trước khi gọi thực hiện constructor, nên constructor không thể là phương thức ảo. Ngược lại do một lớp chỉ có một bảng phương thức ảo nên khi một đối tượng thuộc lớp bị hủy bỏ, bảng phương thức ảo vẫn còn đó, và con trỏ vptr vẫn còn đó. Hơn nữa, destructor được gọi thực hiện trước khi vùng nhớ dành cho đối tượng bị thu hồi, do đó destructor có thể là phương thức ảo. Tuy nhiên, constructor của một lớp có thể gọi phương thức ảo khác. Điều này hoàn toàn không có gì mâu thuẫn với cơ chế kết nối động.

Ví dụ 6.5:

```
1: //Chương trình 6.5: Destructor ảo
2: #include <iostream.h>
3:
4: class Base
5: {
6: public:
7: virtual ~Base()
8: {
9: cout<<"~Base"<<endl;
10: }
11: };
12:
13: class Derived:public Base
14: {
15: public:
16: virtual ~Derived()
17: {
18: cout<<"~Derived"<<endl;
18: }
19: };
20:
21: int main()
22: {
23: Base *B;
24: B = new Derived;
25: delete B;
26: return 0;
27: }
```

Chúng ta chạy ví dụ 6.5, kết quả ở hình 6.6

```
~Derived
~Base
```

Hình 6.6: Kết quả của ví dụ 6.5

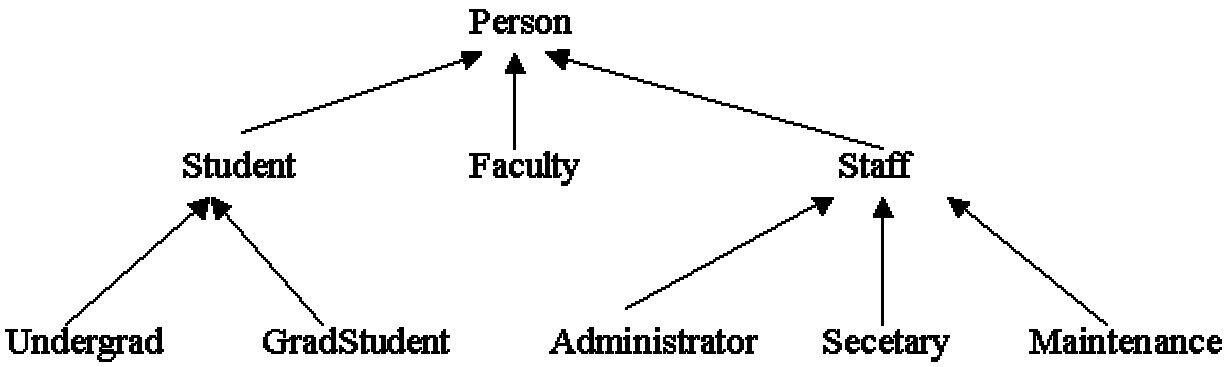
Nếu destructor không là phương thức ảo thì khi giải phóng đối tượng B chỉ có destructor của lớp cơ sơ được gọi mà thôi nhưng khi destructor là phương thức ảo thì khi giải phóng đối tượng B (ở dòng 25) destructor của lớp dẫn xuất được gọi thực hiện rồi đến destructor của lớp cơ sơ.

BÀI TẬP

Bài 1: Hãy xây dựng các lớp cần thiết trong phân cấp hình 5.3 để tính diện tích (hoặc diện tích xung quanh) và thể tích trong đó lớp Shape là lớp cơ sở trừu tượng.

Bài 2: Hãy sửa đổi hệ thống lương của chương trình ở ví dụ 6.6 bằng thêm các thành viên dữ liệu BrithData (một đối tượng kiểu Date) và DepartmentCode (kiểu int) vào lớp Employee. Giả sử lương này được xử lý một lần trên một tháng. Sau đó, chương trình tính bảng lương cho mỗi Employee (tính đa hình), cộng thêm 100.00$ tiền thưởng vào tổng số lương của mỗi người nếu đây là tháng mà ngày sinh của Employee xảy ra.

Bài 3: Cài đặt các lớp trong cây phả hệ lớp sau:



Trong đó các lớp Person, Student và Staff là các lớp trừu tượng, các lớp còn lại là các lớp dẫn xuất thực.

CHƯƠNG 7

THIẾT KẾ CHƯƠNG TRÌNH THEO HƯỚNG ĐỐI TƯỢNG

I. DẪN NHẬP

Trong chương này, chúng ta tìm hiểu một ít về cách thiết kế chương trình theo hướng đối tượng, các bước cơ bản cần thiết khi bắt tay vào viết chương trình trên quan điểm thiết kế và thảo chương.

II. CÁC GIAI ĐOẠN PHÁT TRIỂN HỆ THỐNG

Có năm giai đoạn để phát triển hệ thống phần mềm theo hướng đối tượng:

- Phân tích yêu cầu (Requirement analysis)
- Phân tích (Analysis)
- Thiết kế (Design)
- Lập trình (Programming)
- Kiểm tra (Testing)

- Phân tích yêu cầu

  Bằng việc tìm hiểu các trường hợp sử dụng (use case) để nắm bắt các yêu cầu của khách hàng, của vấn đề cần giải quyết. Qua trường hợp sử dụng này, các nhân tố bên ngoài có tham gia vào hệ thống cũng được mô hình hóa bằng các tác nhân. Mỗi trường hợp sử dụng được mô tả bằng văn bản, đặc tả yêu cầu của khách hàng.

- Phân tích

  Từ các đặc tả yêu cầu trên, hệ thống sẽ bước đầu được mô hình hóa bởi các khái niệm lớp, đối tượng và các cơ chế để diễn tả hoạt động của hệ thống.

  Trong giai đoạn phân tích chúng ta chỉ mô tả các lớp trong lĩnh vực của vấn đề cần giải quyết chứ chúng ta không đi sâu vào các chi tiết kỹ thuật.

- Thiết kế

  Trong giai đoạn thiết kế, các kết quả của quá trình phân tích được mở rộng thành một giải pháp kỹ thuật. Một số các lớp được thêm vào để cung cấp cơ sở hạ tầng kỹ thuật như lớp giao diện, lớp cơ sở dữ liệu, lớp chức năng, …

- Lập trình

  Đây còn gọi là bước xây dựng, giai đoạn này sẽ đặc tả chi tiết kết quả của giai đoạn thiết kế. Các lớp của bước thiết kế sẽ được chuyển thành mã nguồn theo một ngôn ngữ lập trình theo hướng đối tượng nào đó.

- Kiểm tra

  Trong giai đoạn kiểm tra, có bốn hình thức kiểm tra hệ thống:

  - Kiểm tra từng đơn thể (unit testing) được dùng kiểm tra các lớp hoặc các nhóm đơn.
  - Kiểm tra tính tích hợp (integration testing), được kết hợp với các thành phần và các lớp để kiểm tra xem chúng hoạt động với nhau có đúng không.
  - Kiểm tra hệ thống (system testing) chỉ để kiểm tra xem hệ thống có đáp ứng được chức năng mà người dùng yêu cầu không.
  - Kiểm tra tính chấp nhận được(acceptance testing), việc kiểm tra này được thực hiện bởi khách hàng, việc kiểm tra cũng thực hiện giống như kiểm tra hệ thống.

III. CÁCH TÌM LỚP

Lớp nên được tìm từ phạm vi bài toán cần giải quyết, vì vậy tên của lớp cũng nên đặt tên các đối tượng thực mà chúng ta biểu diễn. Để tìm ra lớp cho bài toán, chúng ta cần trả lời các câu hỏi sau:

- Có thông tin nào cần lưu trữ hay phân tích không? Nếu có bất kỳ thông tin nào cần phải lưu trữ, biến đổi, phân tích hoặc xử lý thì đó chính là một lớp dự định cần xây dựng.
- Có hệ thống bên ngoài bên ngoài hay không? Hệ thống ngoài có thể được xem như các lớp mà hệ thống của chúng ta chứa hoặc tương tác với nó.
- Có các mẫu thiết kế, thư viện lớp, thành phần, … hay không? Các thành phần này đã được xây dựng từ các project trước đó, từ các đồng nghiệp hoặc các nhà sản xuất?
- Có thiết bị nào mà hệ thống phải đáp ứng? Bất cứ thiết bị nào được nối với hệ thống có thể chuyển thành lớp dự tuyển.
- Tác nhân đóng vai trò như thế nào trong hệ thống? Các vai diễn này nên được xem là lớp như người sử dụng, khách hang, người điều khiển hệ thống,…

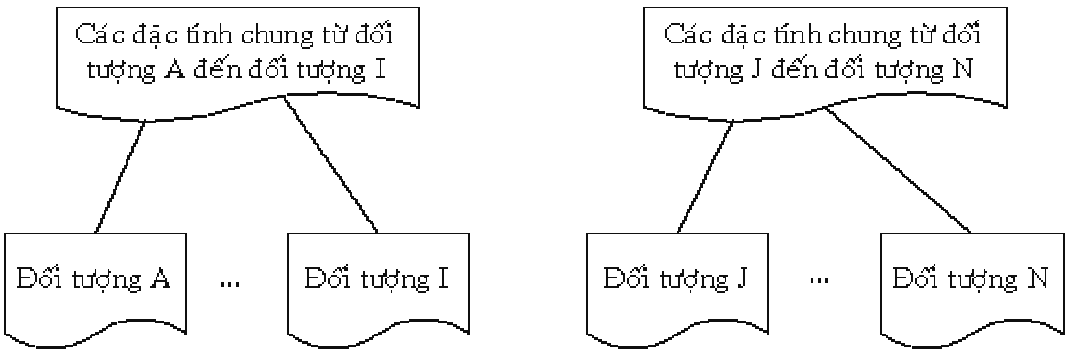
IV. CÁC BƯỚC CẦN THIẾT ĐỂ THIẾT KẾ CHƯƠNG TRÌNH

Để thiết kế một chương trình theo hướng đối tượng, chúng ta phải trải qua bốn bước sau, từ đó chúng ta xây dựng được một cây phả hệ mang tính kế thừa và các mối quan hệ giữa các đối tượng:

- Xác định các dạng đối tượng (lớp) của bài toán (định dang các đối tượng).
- Tìm kiếm các đặc tính chung (dữ liệu chung) trong các dạng đối tượng này, những gì chúng cùng nhau chia xẻ.
- Xác định được lớp cơ sở dựa trên cơ sở các đặc tính chung của các dạng đối tượng.
- Từ lớp cơ sở, sử dụng quan hệ tổng quát hóa để đặc tả trong việc đưa ra các lớp dẫn xuất chứa các thành phần, những đặc tính không chung còn lại của dạng đối tượng. Bên cạnh đó, chúng ta còn đưa ra các lớp có quan hệ với các lớp cơ sở và lớp dẫn xuất; các quan hệ này có thể là quan hệ kết hợp, quan hệ tập hợp lại, quan hệ phụ thuộc.

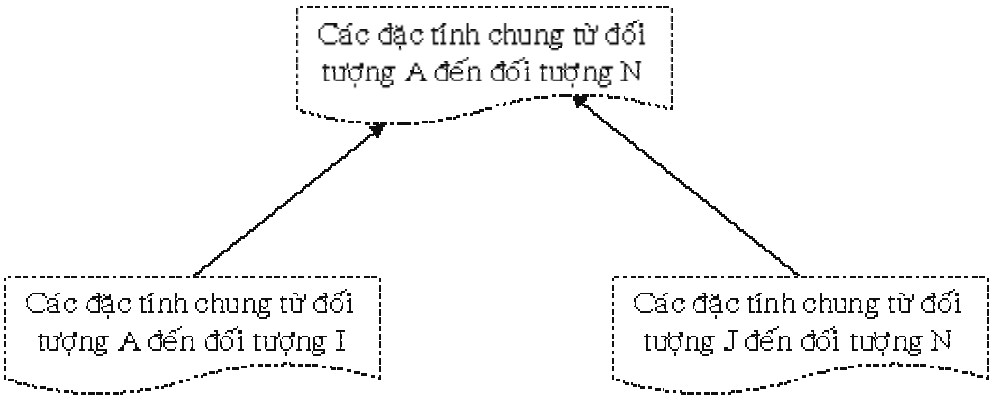
Với các bước trên chúng ta có được cây phả hệ và quan hệ giữa các lớp. Đối với hệ thống phức tạp hơn, chúng ta cần phải phân tích để giải quyết được vấn đề đặt ra theo trật tự sau:

- Phân tích một cách cẩn thận về các đối tượng của bài toán theo trật tự từ dưới lên (bottom up).
- Tìm ra những gì tồn tại chung giữa các đối tượng, nhóm các đặc tính này lại để được các lớp cơ sơ như hình 7.1



Hình 7.1

- Tiếp tục theo hướng từ dưới lên, chúng ta thiết kế được các đối tượng phù hợp như hình 7.2



Hình 7.2

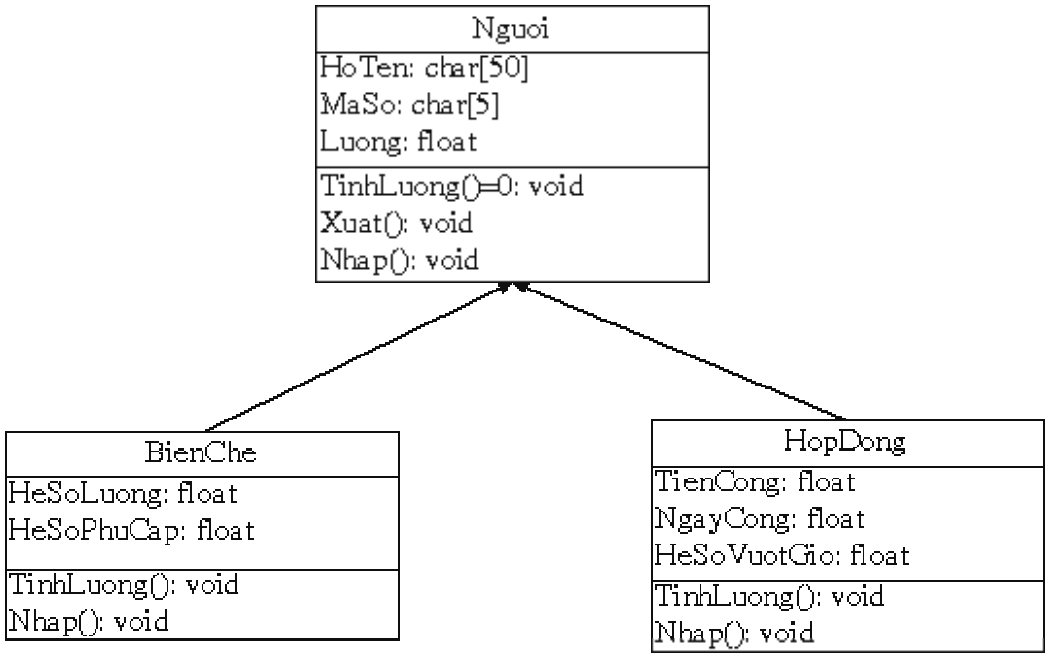
Bằng cách này, chúng ta tiếp tục tìm các đặc tính chung cho đến tột cùng của các đối tượng.

- Sau đó cài đặt theo hướng đối tượng từ trên xuống bằng cách cài đặt lớp cơ sở chung nhất.
- Tiếp tục cài đặt các lớp dẫn xuất trên cơ sở các đặc tính chung của từng nhóm đối tượng.
- Cho đến khi tất cả các dạng đối tượng của hệ thống được cài đặt xong để được cây phả hệ.

V. CÁC VÍ DỤ

Ví dụ 7.1: Tính tiền lương của các nhân viên trong cơ quan theo các dạng khác nhau. Dạng người lao động lãnh lương từ ngân sách Nhà nước được gọi là cán bộ, công chức (dạng biên chế). Dạng người lao động lãnh lương từ ngân sách của cơ quan được gọi là người làm hợp đồng. Như vậy hệ thống chúng ta có hai đối tượng: biên chế và hợp đồng.

- Hai loại đối tượng này có đặc tính chung đó là viên chức làm việc cho cơ quan. Từ đây có thể tạo nên lớp cơ sở để quản lý một viên chức (lớp *Nguoi*) bao gồm mã số, họ tên và lương.
- Sau đó chúng ta xây dựng các lớp còn lại kế thừa từ lớp cơ sở trên:
  - Lớp dành cho cán bộ, công chức (lớp *BienChe*) gồm các thuộc tính: hệ số lương, tiền phụ cấp chức vụ.
  - Lớp dành cho người làm hợp đồng (lớp *HopDong*) gồm các thuộc tính: tiền công lao động, số ngày làm việc trong tháng, hệ số vượt giờ.



Hình 7.3

File PERSON.H

```
1: //PERSON.H
2: Định nghĩa lớp Nguoi
3: #ifndef PERSON_H
4: #define PERSON_H
5:
6: #include <iostream.h>
7:
8: #define MAX_TEN 50
9: #define MAX_MASO 5
10: #define MUC_CO_BAN 120000
11:
12: class Nguoi
13: {
14: protected:
15: char HoTen[MAX_TEN];
16: char MaSo[MAX_MASO];
17: float Luong;
18: public:
19: Nguoi();
20: virtual void TinhLuong()=0;
21: void Xuat() const;
22: virtual void Nhap();
23: };
24:
25: #endif
```

File PERSON.CPP

```
1: //PERSON.CPP
2: Định nghĩa hàm thành viên cho lớp Nguoi
3: #include <iomanip.h>
4: #include <string.h>
5: #include "person.h"
6:
7: Nguoi::Nguoi()
8: {
9: strcpy(HoTen,"");
10: strcpy(MaSo,"");
11: Luong=0;
12: }
13:
14: void Nguoi::Xuat() const
15: {
16: cout<<"Ma so:"<<MaSo<<",Ho va ten:"<<HoTen
17:       #9;
<<",Luong:"<<setiosflags(ios::fixed)<<setprecision(0)<<Luong<<endl;
18: }
19:
20: void Nguoi::Nhap()
21: {
22: cout<<"Ma so:";
23: cin>>MaSo;
24: cin.ignore();
25: cout<<"Ho va ten:";
26: cin.getline(HoTen,MAX_TEN);
27: }
```

File STAFF.H

```
1: //STAFF.H
2 Định nghĩa lớp BienChe
3: #ifndef STAFF_H
4: #define STAFF_H
5:
5: #include "person.h"
6:
7: class BienChe: public Nguoi
8: {
9: protected:
10: float HeSoLuong;
11: float HeSoPhuCap;
12: public:
13: BienChe();
14: virtual void TinhLuong();
15: virtual void Nhap();
16: };
17:
18: #endif
```

File STAFF.CPP

```
1: //STAFF.CPP
2: Định nghĩa hàm thành viên cho lớp BienChe
3: #include "staff.h"
4:
5: BienChe::BienChe()
6: {
7: HeSoLuong=HeSoPhuCap=0;
8: }
9:
10: void BienChe::Nhap()
11: {
12: Nguoi::Nhap();
13: cout<<"He so luong:";
14: cin>>HeSoLuong;
15: cout<<"He so phu cap chu vu:";
16: cin>>HeSoPhuCap;
17: }
18:
19: void BienChe::TinhLuong()
20: {
21: Luong=MUC_CO_BAN*(1.0+HeSoLuong+HeSoPhuCap);
22: }
```

File CONTRACT.H

```
1: //CONTRACT.H
2: Định nghĩa lớp HopDong
3: #ifndef CONTRACT_H
4: #define CONTRACT_H
5:
6: #include "person.h"
7:
8: class HopDong : public Nguoi
9: {
10: protected:
11: float TienCong;
12: float NgayCong;
```

```
13: float HeSoVuotGio;
14: public:
15: HopDong();
16: virtual void TinhLuong();
17: virtual void Nhap();
18: };
19:
20: #endif
```

File CONTRACT.CPP:

```
1: //CONTRACT.CPP
2: Định nghĩa hàm thành viên cho lớp HopDong
3: #include "contract.h"
4:
5: HopDong::HopDong()
6: {
7: TienCong=NgayCong=HeSoVuotGio=0;
8: }
9:
10: void HopDong::Nhap()
11: {
12: Nguoi::Nhap();
13: cout<<"Tien cong:";
14: cin>>TienCong;
15: cout<<"Ngay cong:";
16: cin>>NgayCong;
17: cout<<"He so vuot gio:";
18: cin>>HeSoVuotGio;
19: }
20:
21: void HopDong::TinhLuong()
22: {
23: Luong=TienCong*NgayCong*(1+HeSoVuotGio);
24: }
```

File CT7_1.CPP:

```
1: //CT7_1.CPP
2: //Chương trình 7.1
3: #include <iostream.h>
4: #include <ctype.h>
5: #include "person.h"
6: #include "staff.h"
7: #include "contract.h"
8:
9: int main()
10: {
11: Nguoi *Ng[100];
12: int N=0;
13: char Chon,Loai;
14: do
15: {
16: cout<<"Bien che hay Hop dong (B/H)? ";
17: cin>>Loai;
18: Loai=toupper(Loai);
19: if (Loai=='B')
20: Ng[N]=new BienChe;
21: else
22: Ng[N]=new HopDong;
```

```
23: Ng[N++]->Nhap();
24: cout<<"Tiep tuc (C/K)? ";
25: cin>>Chon;
26: Chon=toupper(Chon);
27: if ((N==100)||(Chon=='K'))
28: break;
29: }
30: while (1);
31: for(int I=0;I<N;++I)
32: {
33: Ng[I]->TinhLuong();
34: Ng[I]->Xuat();
35: }
36: return 0;
37: }
```

Chúng ta chạy ví dụ 7.1, kết quả ở hình 7.4

```
Bien che hay Hop dong (B/H)? b
Ma so: 1000
Ho va ten: Tran Van Binh
He so luong: 0.7
He so phu cap chu vu: 0
Tiep tuc (C/K)? c
Bien che hay Hop dong (B/H)? h
Ma so: 1001
Ho va ten: Nguyen Van Tuan
Tien cong: 5000
Ngay cong: 20
He so vuot gio: 0
Tiep tuc (C/K)? k
Ma so: 1000,Ho va ten: Tran Van Binh,Luong: 204000
Ma so: 1001,Ho va ten: Nguyen Van Tuan,Luong: 100000
```

Hình 7.4: Kết quả của ví dụ 7.1

Ví dụ 7.2: Giả sử cuối năm học cần trao giải thưởng cho các sinh viên xuất sắc và các giảng viên có nhiều công trình khoa học được công bố trên tạp chí. Các lớp trong cây phả hệ như hình 7.5: lớp *Nguoi* để quản lý hồ sơ cá nhân, lớp *SinhVien* quản lý về sinh viên và lớp *GiangVien* quản lý giảng viên.

Lớp *Nguoi*:

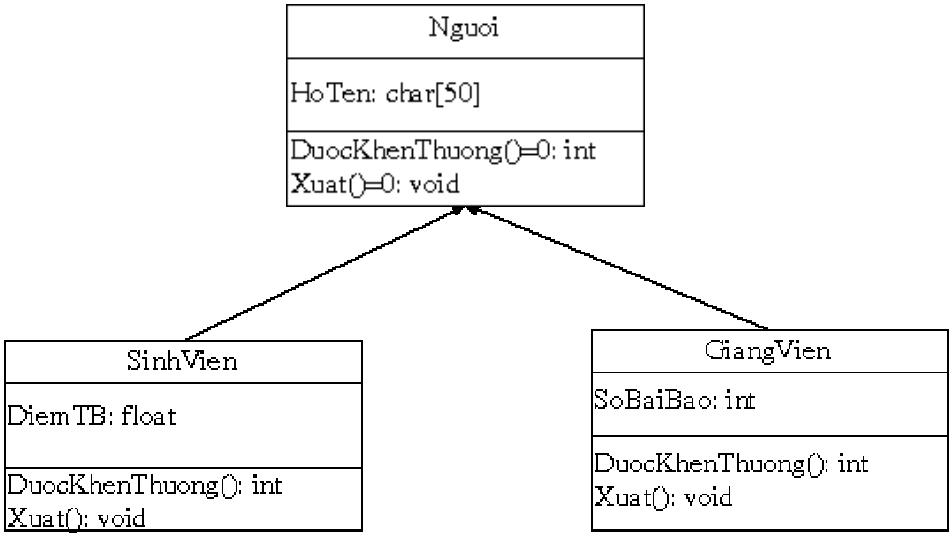
- Dữ liệu họ và tên.
- Phương thức kiểm tra khả năng được khen thưởng. Đây là phương thức thuần ảo.
- Phương thức xuất. Đây là phương thức thuần ảo.

Lớp *SinhVien*:

- Dữ liệu điểm trung bình.
- Phương thức kiểm tra khả năng được khen thưởng.
- Phương thức xuất.

Lớp *GiangVien*:

- Dữ liệu điểm trung bình.
- Phương thức kiểm tra khả năng được khen thưởng.
- Phương thức xuất.



Hình 7.5

File PERSON.H

```
1: //PERSON.H
2: Định nghĩa lớp Nguoi
3: #ifndef PERSON_H
4: #define PERSON_H
5:
6: #include <iostream.h>
7:
8: #define MAX_TEN 50
9:
10: class Nguoi
11: {
12: protected:
13: char HoTen[MAX_TEN];
14: public:
15: Nguoi(char *HT);
16: virtual int DuocKhenThuong() const=0;
17: virtual void Xuat() const=0;
18: };
19:
20: #endif
```

File PERSON.CPP:

```
1: //PERSON.CPP
2: Định nghĩa hàm thành viên cho lớp Nguoi
3: #include <string.h>
4: #include "person.h"
5:
6: Nguoi::Nguoi(char *HT)
7: {
8: strcpy(HoTen,HT);
9: }
```

File STUDENT.H:

```
1: //STUDENT.H
2: Định nghĩa lớp SinhVien
3: #ifndef STUDENT_H
4: #define STUDENT_H
5:
```

```
6: #include "person.h"
7:
8: class SinhVien : public Nguoi
9: {
10: protected:
11: float DiemTB;
12: public:
13: SinhVien(char *HT,float DTB);
14: virtual int DuocKhenThuong() const;
15: virtual void Xuat() const;
16: };
17:
18: #endif
```

File STUDENT.CPP:

```
1: //STUDENT.CPP
2: Định nghĩa hàm thành viên cho lớp SinhVien
3: #include "student.h"
4:
5: SinhVien::SinhVien(char *HT,float DTB):Nguoi(HT)
6: {
7: DiemTB=DTB;
8: }
9:
10: int SinhVien::DuocKhenThuong() const
11: {
12: return DiemTB>9.0;
13: }
14:
15: void SinhVien::Xuat() const
16: {
17: cout<<"Ho va ten cua sinh vien:"<<HoTen;
18: }
```

File TEACHER.H:

```
1: //TEACHER.H
2: Định nghĩa lớp GiangVien
3: #ifndef TEACHER_H
4: #define TEACHER_H
5:
6: #include "person.h"
7:
8: class GiangVien : public Nguoi
9: {
10: protected:
11: int SoBaiBao;
12: public:
13: GiangVien(char *HT,int SBB);
14: virtual int DuocKhenThuong() const;
15: virtual void Xuat() const;
16: };
17:
18: #endif
```

File TEACHER.CPP:

```
1: //TEACHER.CPP
2: Định nghĩa hàm thành viên cho lớp GiangVien
3: #include "teacher.h"
4:
```

```
5: GiangVien::GiangVien(char *HT,int SBB):Nguoi(HT)
6: {
7: SoBaiBao=SBB;
8: }
9:
10: int GiangVien::DuocKhenThuong() const
11: {
12: return SoBaiBao>5;
13: }
14:
15: void GiangVien::Xuat() const
16: {
17: cout<<"Ho va ten cua giang vien:"<<HoTen;

18: }
```

File CT7_2.CPP:

```
1: //CT7_2.CPP
2: //Chương trình 7.2
3: #include <ctype.h>
4: #include "person.h"
5: #include "student.h"
6: #include "teacher.h"
7:
8: int main()
9: {
10: Nguoi *Ng[100];
11: int N=0;
12: char Chon,Loai;
13: char HoTen[MAX_TEN];
14: do
15: {
16: cout<<"Ho va ten:";
17: cin.getline(HoTen,MAX_TEN);
18: cout<<"Sinh vien hay Giang vien(S/G)? ";
19: cin>>Loai;
20: Loai=toupper(Loai);
21: if (Loai=='S')
22: {
23: float DTB;
24: cout<<"Diem trung binh:";
25: cin>>DTB;
26: Ng[N++]=new SinhVien(HoTen,DTB);
27: }
28: else
29: {
30: int SoBaiBao;
31: cout<<"So bai bao:";
32: cin>>SoBaiBao;
33: Ng[N++]=new GiangVien(HoTen,SoBaiBao);
34: }
35: cout<<"Tiep tuc (C/K)? ";
36: cin>>Chon;
37: Chon=toupper(Chon);
38: cin.ignore();
39: if ((N==100)||(Chon=='K'))
40: break;
41: }
42: while (1);
```

```
43: for(int I=0;I<N;++I)
44: {
45: Ng[I]->Xuat();
46: if (Ng[I]->DuocKhenThuong())
47: cout<<". Nguoi nay duoc khen thuong";
48: cout<<endl;
49: }
50: return 0;
51: }
```

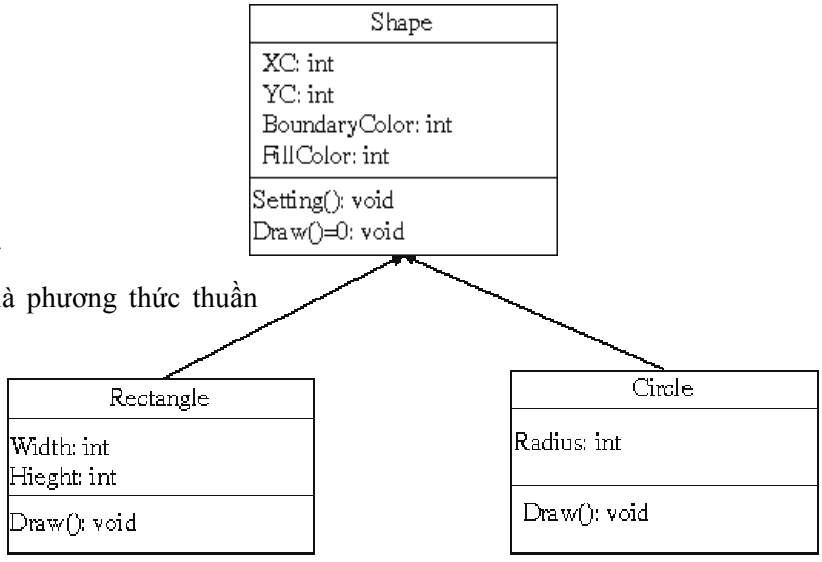
Chúng ta chạy ví dụ 7.2, kết quả ở hình 7.6

```
Ho va ten:Nguyen Van Hieu
Sinh vien hay Giang vien(S/G)? s
Diem trung binh:8
Tiep tuc (C/K)? c
Ho va ten:Tran Van Truong
Sinh vien hay Giang vien(S/G)? s
Diem trung binh:9.5
Tiep tuc (C/K)? c
Ho va ten:Nguyen Phuoc Trong
Sinh vien hay Giang vien(S/G)? g
So bai bao:4
Tiep tuc (C/K)? c
Ho va ten:Pham Xuan Minh
Sinh vien hay Giang vien(S/G)? g
So bai bao:6
Tiep tuc (C/K)? k
Ho va ten cua sinh vien:Nguyen Van Hieu
Ho va ten cua sinh vien:Tran Van Truong. Nguoi nay duoc khen thuong
Ho va ten cua giang vien:Nguyen Phuoc Trong
Ho va ten cua giang vien:Pham Xuan Minh. Nguoi nay duoc khen thuong
```

Hình 7.6: Kết quả của ví dụ 7.2

Ví dụ 7.3: Giả sử cần phải tạo các hình: hình tròn và hình chữ nhật được tô theo hai màu red và blue. Xây dựng một cây phả hệ để quản lý các hình này.

Trước hết chúng ta cần có lớp cơ sở *Shape* để lưu trữ thông tin chung cho các hình, sau đó là hai lớp dẫn xuất *Rectangle* về hình hình chữ nhật và *Circle* về hình tròn như hình 7.7



Lớp *Shape*:

- Tọa độ tâm.
- Màu đường biên.
- Màu tô.
- Phương thức thiết lập tô màu.
- Phương thức vẽ hình. Đây là phương thức thuần ảo.

Lớp *Rectangle*:

- Chiều dài và chiều rộng.
- Phương thức vẽ hình.

Lớp *Circle*:

- Bán kính.
- Phương thức vẽ hình.

CHƯƠNG 8

CÁC DẠNG NHẬP/XUẤT

I. DẪN NHẬP

Các thư viện chuẩn C++ cung cấp một tập hợp các khả năng nhập/xuất rộng lớn. Trong chương này chúng ta tìm hiểu một phạm vi của các khả năng đủ để phần lớn các thao tác nhập xuất.

Phần lớn các đặc tính nhập xuất mô tả ở đây theo hướng đối tượng. Kiểu này của nhập/xuất thi hành việc sử dụng các đặc tính khác của C++ như các tham chiếu, đa năng hóa hàm và đa năng hóa toán tử.

Như chúng ta sẽ thấy, C++ sử dụng nhập/xuất kiểu an toàn (type safe). Mỗi thao tác nhập/xuất được thực hiện một cách tự động theo lối nhạy cảm về kiểu dữ liệu. Mỗi thao tác nhập xuất có được định nghĩa thích hợp để xử lý một kiểu dữ liệu cụ thể thì hàm đó được gọi để xử lý kiểu dữ liệu đó. Nếu không có đối sánh giữa kiểu của dữ liệu hiện tại và một hàm cho việc xử lý kiểu dữ liệu đó, một chỉ dẫn lỗi biên dịch được thiết lập. Vì thế dữ liệu không thích hợp không thể "lách" qua hệ thống.

Các người dùng có thể chỉ định nhập/xuất của các kiểu dữ liệu do người dùng định nghĩa cũng như các kiểu dữ liệu chuẩn. Tính mở rộng này là một trong các đặc tính quan trọng của C++.

II. CÁC DÒNG(STREAMS)

Nhập/xuất C++ xảy ra trong các dòng của các byte. Một dòng đơn giản là một dãy tuần tự các byte. Trong các thao tác nhập, các byte chảy từ thiết bị (chẳng hạn: một bàn phím, một ổ đĩa, một kết nối mạng) tới bộ nhớ chính. Trong các thao tác xuất, các byte chảy từ bộ nhớ chính tới một thiết bị (chẳng hạn: một màn hình, một máy in, một ổ đĩa, một kết nối mạng).

Ưùng dụng liên kết với các byte. Các byte có thể biểu diễn các ký tự ASCII, bên trong định dạng dữ liệu thô, các ảnh đồ họa, tiếng nói số, hình ảnh số hoặc bất cứ loại thông tin một ứng dụng có thể đòi hỏi.

Công việc của các cơ chế hệ thống nhập/xuất là di chuyển các byte từ các thiết bị tới bộ nhớ và ngược lại theo lối chắc và đáng tin cậy. Như thế các di chuyển thường bao gồm sự di chuyển cơ học như sự quay của một đĩa hoặc một băng từ, hoặc nhấn phím tại một bàn phím. Thời gian các di chuyển này thông thường khổng lồ so với thời gian bộ xử lý thao tác dự liệu nội tại. Vì thế, các thao tác nhập/xuất đòi hỏi có kế hoạch cẩn thận và điều chỉnh để bảo đảm sự thi hành tối đa.

C++ cung cấp cả hai khả năng nhập/xuất "mức thấp" (low-level) và "mức cao" (high-level). Các khả năng nhập/xuất mức thấp (nghĩa là nhập/xuất không định dạng) chỉ định cụ thể số byte nào đó phải được di chuyển hoàn toàn từ thiết bị tới bộ nhớ hoặc từ bộ nhớ tới thiết bị. Trong các di chuyển như thế, byte riêng rẽ là mục cần quan tâm. Vì thế các khả năng mức thấp cung cấp tốc độ cao, các di chuyển dung lượng cao, nhưng các khả năng này không phải là tiện lợi lắm cho lập trình viên.

Các lập trình viên ưu thích quan điểm nhập/xuất mức cao, nghĩa là nhập/xuất có định dạng, trong đó các byte được nhóm thành các đơn vị có ý nghĩa như các số nguyên, các số chấm động, các ký tự, các chuỗi và các kiểu do người dùng định nghĩa.

II.1. Các file header của thư viện iostream

- Thư viện **iostream** của C++ cung cấp hàng trăm khả năng của nhập/xuất. Một vài tập tin header chứa các phần của giao diện thư viện.

- Phần lớn chương trình C++ thường include tập tin header **<iostream.h>** mà chứa các thông tin cơ bản đòi hỏi tất cả các thao tác dòng nhập/xuất. Tập tin header **<iostream.h>** chứa các đối tượng **cin**, **cout**, **cerr** và **clog** mà tương ứng với dòng nhập chuẩn, dòng xuất chuẩn, dòng lỗi chuẩn không vùng đệm và dòng lỗi chuẩn vùng đệm. Cả hai khả năng nhập/xuất định dạng và không định dạng được cung cấp.

- Header **<iomanip.h>** chứa thông tin hữu ích cho việc thực hiện nhập/xuất định dạng với tên gọi là các bộ xử lý dòng biểu hiện bằng tham số (parameterized stream manipulators).

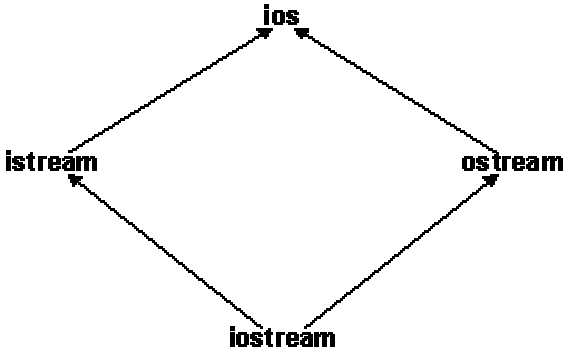
- Header **<fstream.h>** chứa các thông tin quan trọng cho các thao tác xử lý file do người dùng kiểm soát.

- Header **<strstream.h>** chứa các thông tin quan trọng cho việc thực hiện các định dạng trong bộ nhớ. Điều này tương tự xử lý file, nhưng các thao tác nhập/xuất tới và từ mảng các ký tự hơn là file.

- Header **<stdiostream.h>** chứa các thông tin quan trọng cho các chương trình trộn các kiểu nhập/xuất của C và C++. Các chương trình mới phải tránh kiểu nhập/xuất C, nhưng cần thì hiệu chỉnh các chương trình C, hoặc tiến triển chương trình C thành C++.

II.2. Các lớp và các đối tượng của dòng nhập/xuất

Thư viện **iostream** chứa nhiều lớp để xử lý một sự đa dạng rộng của các thao tác nhập/xuất. Lớp **istream** hỗ trợ các thao tác dòng nhập. Lớp ostream hỗ trợ các thao tác dòng xuất. Lớp **iostream** hỗ trợ cả hai thao tác dòng nhập và dòng xuất. Lớp **istream** và lớp **ostream** đều kế thừa đơn từ lớp cơ sở **ios**. Lớp **iostream** được kế thừa thông qua đa kế thừa từ hai lớp **istream** và **ostream**.



Hình 8.1: Một phần của phân cấp lớp dòng nhập/xuất

Đa năng hóa toán tử cung cấp một ký hiệu thích hợp cho việc thực hiện nhập/xuất. Toán tử dịch chuyển trái (<<) được đa năng hóa để định rõ dòng xuất và được tham chiếu như là toán tử chèn dòng. Toán tử dịch chuyển phải (>>) được đa năng hóa để định rõ dòng nhập và được tham chiếu như là toán tử trích dòng. Các toán tử này được sử dụng với các đối tượng dòng chuẩn **cin**, **cout**, **cerr** và **clog**, và bình thường với các đối tượng dòng do người dùng định nghĩa.

- **cin** là một đối tượng của lớp **istream** và được nói là "bị ràng buộc tới" (hoặc kết nối tới) thiết bị nhập chuẩn, thông thường là bàn phím. Toán tử trích dòng được sử dụng ở lệnh sau tạo ra một giá trị cho biến nguyên *X* được nhập từ **cin** tới bộ nhớ:

```
int X;
cin >> X;
```

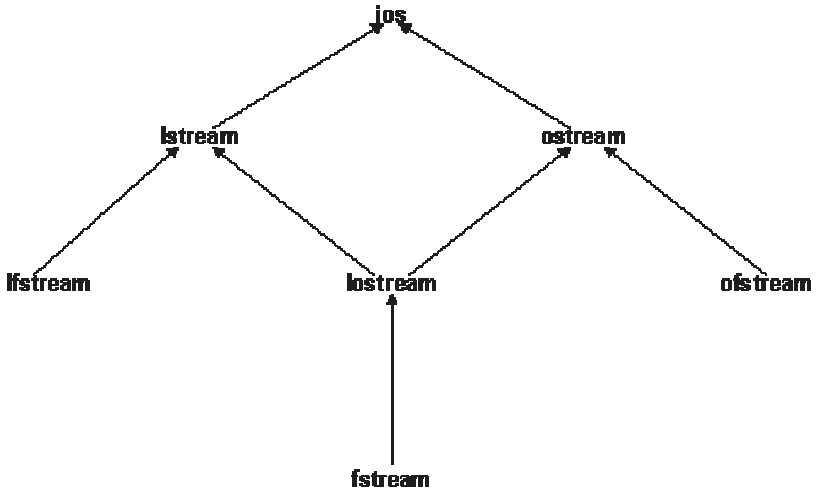
- **cout** là một đối tượng của lớp **ostream** và được nói là "bị ràng buộc tới" thiết bị xuất chuẩn, thông thường là màn hình. Toán tử chèn dòng được sử dụng ở lệnh sau tạo ra một giá trị cho biến nguyên *X* được xuất từ bộ nhớ tới thiết bị chuẩn:

```
cout << X;
```

- **cerr** là một đối tượng của lớp **ostream** và được nói là "bị ràng buộc tới" thiết bị lỗi chuẩn. Việc xuất đối tượng **cerr** là không vùng đệm. Điều này có nghĩa là mỗi lần chèn tới **cerr** tạo ra kết xuất của nó xuất hiện ngay tức thì; Điều này thích hợp cho việc thông báo nhanh chóng người dùng khi có sự cố.

- **clog** là một đối tượng của lớp **ostream** và được nói là "bị ràng buộc tới" thiết bị lỗi chuẩn. Việc xuất đối tượng **cerr** là có vùng đệm. Điều này có nghĩa là mỗi lần chèn tới **cerr** tạo ra kết xuất của nó được giữ trong vùng đệm cho đến khi vùng đệm đầy hoặc vùng đệm được flush.

Việc xử lý file của C++ sử dụng các lớp **ifstream** để thực hiện các thao tác nhập file, **ofstream** cho các thao tác xuất file, và **fstream** cho các thao tác nhập/xuất file. Lớp **ifstream** kế thừa từ **istream**, **ofstream** lớp kế thừa từ **ostream**, và lớp **fstream** kế thừa từ **iostream**.



Hình 8.2: Một phần của phân cấp lớp dòng nhập/xuất với việc xử lý file.

III. DÒNG XUẤT

ostream của C++ cung cấp khả năng để thực hiện xuất định dạng và không định dạng. Các khả năng xuất bao gồm: xuất các kiểu dữ liệu chuẩn với toán tử chèn dòng; xuất các ký tự với hàm thành viên **put()**; xuất không định dạng với hàm thành viên **write**; xuất các số nguyên dạng thập phân, bát phân và thập lục phân; xuất các giá trị chấm động với độ chính xác khác nhau, với dấu chấm thập phân, theo ký hiệu khoa học và theo ký hiệu cố định; xuất dữ liệu theo các trường độn thêm các ký tự chỉ định; và xuất các mẫu tự chữ hoa theo ký hiệu khoa học và ký hiệu thập lục phân.

III.1. Toán tử chèn dòng

Dòng xuất có thể được thực hiện với toán tử chèn dòng, nghĩa là toán tử << đã đa năng hóa. Toán tử << đã được đa năng hóa để xuất các mục dữ liệu của các kiểu có sẵn, xuất chuỗi, và xuất các giá trị con trỏ.

Ví dụ 8.1: Minh họa xuất chuỗi sử dụng một lệnh chèn dòng.

```
1: //Chương trình 8.1:Xuất một chuỗi sử dụng chèn dòng
2: #include <iostream.h>
3:
4: int main()
5: {
6: cout<<"Welcome to C++!\n";
7: return 0;
8: }
```

Chúng ta chạy ví dụ 8.1, kết quả ở hình 8.3

Hình 8.3: Kết quả của ví dụ 8.1

Ví dụ 8.2: Minh họa xuất chuỗi sử dụng nhiều lệnh chèn dòng.

```
1: //Chương trình 8.2:Xuất một chuỗi sử dụng hai chèn dòng
2: #include <iostream.h>
3:
4: int main()
5: {
6: cout<<"Welcome to";
7: cout<<"C++!\n";
8: return 0;
9: }
```

Chúng ta chạy ví dụ 8.2, kết quả ở hình 8.4

Hình 8.4: Kết quả của ví dụ 8.2

Hiệu quả của chuỗi thoát **\n** (newline) cũng đạt được bởi bộ xử lý dòng (stream manipulator) **endl** (end line).

Ví dụ 8.3:

```
1: //Chương trình 8.3:Sử dụng bộ xử lý dòng endl
2: #include <iostream.h>
3:
4: int main()
5: {
6: cout<<"Welcome to";
7: cout<<"C++!";
8: cout<<endl;
9: return 0;
10: }
```

Chúng ta chạy ví dụ 8.3, kết quả ở hình 8.5

Hình 8.5: Kết quả của ví dụ 8.3

Bộ xử lý dòng **endl** đưa ra một ký tự newline, và hơn nữa, flush vùng đệm xuất (nghĩa là tạo ra vùng đệm xuất được xuất ngay lập tức kể cả nó chưa đầy). Vùng đệm xuất cũng có thể được flush bằng:

```
cout<<flush;
```

Ví dụ 8.4: Các biểu thức có thể xuất

```
1: //Chương trình 8.4: Xuất giá trị biểu thức.
2: #include <iostream.h>
3:
4: int main()
5: {
6: cout<<"47 plus 53 is ";
7: cout<< (47+53);
8: cout<<endl;
9: return 0;
10: }
```

Chúng ta chạy ví dụ 8.4, kết quả ở hình 8.6

Hình 8.6: Kết quả của ví dụ 8.4

III.2. Nối các toán tử chèn dòng và trích dòng

Các toán tử đã đa năng hóa **<<** và **>>** có thể được theo dạng nối vào nhau.

Ví dụ 8.5: Nối các toán tử đã đa năng hóa

```
1: //Chương trình 8.5: Nối toán tử << đã đa năng hóa.
2: #include <iostream.h>
3:
4: int main()
5: {
```

```
6: cout<<"47 plus 53 is "<< (47+53)<<endl;
7: return 0;
8: }
```

Chúng ta chạy ví dụ 8.5, kết quả ở hình 8.7

Hình 8.7: Kết quả của ví dụ 8.5

Nhiều chèn dòng ở dòng 6 trong ví dụ 8.5 được thực thi nếu có thể viết:

(((cout<<"47 plus 53 is ")<< (47+53))<<endl);

nghĩa là **<<** liên kết từ trái qua phải. Loại liên kết của các toán tử chèn dòng được phép bởi vì toán tử đa năng hóa **<<** trả về một tham chiếu tới đối tượng toán hạng bên trái của nó, nghĩa là **cout**. Vì thế biểu thức đặt trong ngoặc bên cực trái:

(cout<<"47 plus 53 is ")

xuất ra một chuỗi đã chỉ định và trả về một tham chiếu tới **cout**. Điều này cho phép biểu thức đặt trong ngoặc ở giữa được ước lượng:

(cout<< (47+53))

xuất giá trị nguyên 100 và trả về một tham chiếu tới **cout**. Sau đó biểu thức đặt trong ngoặc bên cực phải được ước lượng:

cout<<endl;

xuất một newline, flush cout và trả về một tham chiếu tới **cout**. Trả về cuối cùng này không được sử dụng.

8.3.3 Xuất các biến kiểu char *:

Trong nhập/xuất kiểu C, thật cần thiết cho lập trình viên để cung cấp thông tin kiểu. C++ xác định các kiểu dữ liệu một cách tự động – một cải tiến hay hơn C. Đôi khi điều này là một trở ngại. Chẳng hạn, chúng ta biết rằng một chuỗi ký tự là kiểu char *. Mục đích của chúng ta in giá trị của con trỏ đó, nghĩa là địa chỉ bộ nhớ của ký tự đầu tiên của chuỗi đó. Nhưng toán tử << đã được đa năng hóa để in dữ liệu của kiểu char * như là chuỗi kết thúc ký tự null. Giải pháp là ép con trỏ thành kiểu void *.

Ví dụ 8.6: In địa chỉ lưu trong một biến kiểu char *

Chúng ta chạy ví dụ 8.6, kết quả ở hình 8.8

Hình 8.8: Kết quả của ví dụ 8.6

III.3. Xuất ký tự với hàm thành viên put(); Nối với nhau hàm put()

Hàm thành viên **put()** của lớp **ostream** xuất một ký tự có dạng :

ostream& put(char *ch***);**

Chẳng hạn:

cout.put('A');

Gọi **put()** có thể được nối vào nhau như:

```
cout.put('A').put('\n');
```

Hàm **put()** cũng có thể gọi với một biểu thức có giá trị là mã ASCII như:

```
cout.put(65);
```

IV. DÒNG NHẬP

Dòng nhập có thể được thực hiện với toán tử trích, nghĩa là toán tử đã đa năng hóa >>. Bình thường toán tử này bỏ qua các ký tử khoảng trắng (như các blank, tab và newline). trong dòng nhập. Toán tử trích dòng trả về zero (false) khi kết thúc file (end-of-file) được bắt gặp trên một dòng; Ngược lại, toán tử trích dòng trả về một tham chiếu tới đối tượng xuyên qua đó nó được kéo theo. Mỗi dòng chứa một tập các bit trạng thái (state bit) sử dụng để điều khiển trạng thái của dòng (nghĩa là định dạng, ấn định các trạng thái lỗi,…). Trích dòng sinh ra **failbit** của dòng được thiết lập nếu dữ liệu của kiểu sai được nhập, và sinh ra **badbit** của dòng được thiết lập nếu thao tác sai.

IV.1. Toán tử trích dòng:

Để đọc hai số nguyên sử dụng đối tượng **cin** và toán tử trích dòng đã đa năng hóa **>>.**

Ví dụ 8.7:

```
1: //Chương trình 8.7
2: #include <iostream.h>
3:
4: int main()
5: {
6: int X, Y;
7: cout << "Enter two integers: ";
8: cin >> X >> Y;
9: cout << "Sum of " << X << " and " << Y << " is: "
10:       << (X + Y) << endl;
11: return 0;
12: }
```

Chúng ta chạy ví dụ 8.7, kết quả ở hình 8.9

```
Enter two integers: 30 92
Sum of 30 and 92 is: 122
```

Hình 8.9: Kết quả của ví dụ 8.7

Một cách phổ biến để nhập một dãy các giá trị là sử dụng toán tử trích dòng trong vòng lặp **while**. Toán tử trích dòng trả về false (0) khi end-of-file được bắt gặp:

Ví dụ 8.8:

```
2: #include <iostream.h>
3:
4: int main()
5: {
6: int Grade, HighestGrade = -1;
7: cout << "Enter grade (enter end-of-file to end): ";
8: while (cin >> Grade)
9: {
10: if (Grade > HighestGrade )
11: HighestGrade = Grade;
12: cout << "Enter grade (enter end-of-file to end): ";
13: }
14: cout <<endl<< "Highest grade is:" << HighestGrade<< endl;
15: return 0;
16: }
```

Chúng ta chạy ví dụ 8.8, kết quả ở hình 8.10

```
Enter grade (enter end-of-file to end): 67
Enter grade (enter end-of-file to end): 87
Enter grade (enter end-of-file to end): 73
Enter grade (enter end-of-file to end): 95
Enter grade (enter end-of-file to end): 34
Enter grade (enter end-of-file to end): 99
Enter grade (enter end-of-file to end): ^Z

Highest grade is: 99
```

Hình 8.10: Kết quả của ví dụ 8.8

IV.2. Các hàm thành viên get() và getline()

Hàm **istream::get()** có các dạng sau:

(1) **int get();**

(2) **istream& get(unsigned char &*ch*);**

(3) **istream& get(signed char &*ch*);**

(4) **istream& get(unsigned char * *puch*, int *len*, char *delim*='\n');**

(5) **istream& get(signed char * *psch*, int *len*, char *delim*='\n');**

Dạng (1) trích ký tự đơn từ dòng và trả về nó hoặc **EOF** khi end-of-file trên dòng được bắt gặp.

Dạng (2) và (3) trích một ký tự đơn từ dòng và lưu trữ nó vào *ch*.

Dạng (4) và (5) trích các ký tự từ dòng cho đến khi hoặc *delim* được tìm thấy, giới hạn *len* đạt đến, hoặc end-of-file được bắt gặp. Các ký tự được lưu trong con trỏ chỉ đến mảng ký tự *puch* hoặc *psch*.

Ví dụ 8.9: Sử dụng hàm **get()** dạng (1)

```
1: //Chương trình 8.9
2: #include <iostream.h>
3: int main()
4: {
5: int Ch;
6: cout << "Before input, cin.eof() is " << cin.eof() << endl
7:          << "Enter a sentence followed by end-of-file:" << endl;
8: while ( ( Ch = cin.get() ) != EOF)
9: cout.put(Ch);
10: cout << endl << "EOF in this system is: " << Ch << endl;
11: cout << "After input, cin.eof() is " << cin.eof() << endl;
12: return 0;
13: }
```

Chúng ta chạy ví dụ 8.9, kết quả ở hình 8.11

```
Before input, cin.eof() is 0
Enter a sentence followed by end-of-file:
Testing the get and put member functions^Z
Testing the get and put member functions
EOF in this system is: -1
After input, cin.eof() is 1
```

Hình 8.11: Kết quả của ví dụ 8.9

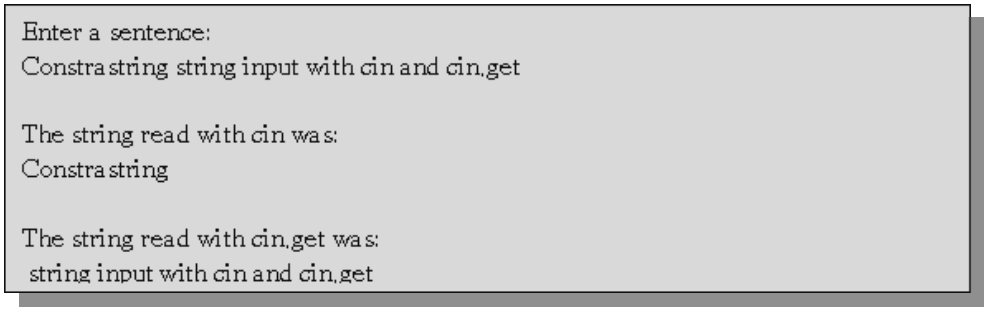
Trong ví dụ 8.9 trên, chúng ta có sử dụng hàm **ios::eof()** có dạng sau:

int eof(); Hàm trả về giá tri khác zero nếu end-of-file bắt gặp.

Ví dụ 8.10: Sử dụng hàm **get()** dạng (5)

```
1: //Chương trình 8.10
2: #include <iostream.h>
3:
4: const int SIZE = 80;
5:
6: int main()
7: {
8: char Buffer1[SIZE], Buffer2[SIZE];
9: cout << "Enter a sentence:" << endl;
10: cin >> Buffer1;
11: cout << endl << "The string read with cin was:" << endl
12:          << Buffer1 << endl << endl;
13: cin.get(Buffer2, SIZE);
14: cout << "The string read with cin.get was:" << endl
15:          << Buffer2 << endl;
16: return 0;
17: }
```

Chúng ta chạy ví dụ 8.10, kết quả ở hình 8.12



Hình 8.12: Kết quả của ví dụ 8.10

Hàm **istream::getline()** có các dạng sau:

(1) **istream& getline(unsigned char *** *puch*, **int** *len*, **char** *delim*=**'\n');**

(2) **istream& getline(signed char *** *psch*, **int** *len*, **char** *delim*=**'\n');**

Ví dụ 8.11: Sử dụng hàm **getline()**

```
1: //Chương trình 8.11
2: #include <iostream.h>
3:
4: const SIZE = 80;
5:
6: int main()
7: {
8: char Buffer[SIZE];
9: cout << "Enter a sentence:" << endl;
10: cin.getline(Buffer, SIZE);
11: cout << endl << "The sentence entered is:" << endl
12:          << Buffer << endl;
13: return 0;
14: }
```

Chúng ta chạy ví dụ 8.11, kết quả ở hình 8.13

```
Enter a sentence:
Using the getline member function

The sentence entered is:
Using the getline member function
```

Hình 8.13: Kết quả của ví dụ 8.11

IV.3. Các hàm thành viên khác của istream

Hàm **ignore()**:

istream& ignore(int *nCount* **= 1, int** *delim* **= EOF);**

Trích và loại bỏ lên đến *nCount* ký tự. Việc trích dừng nếu *delim* được bắt gặp hoặc nếu end-of-file bắt gặp.

Hàm **putback()**:

istream& putback(char *ch***);**

Đặt một ký tự ngược lại dòng nhập.

Hàm **peek()**:

int peek();

Hàm trả về ký tự kế tiếp mà không trích nó từ dòng.

IV.4. Nhập/xuất kiểu an toàn

C++ cung cấp nhập/xuất kiểu an toàn (type-safe). Các toán tử << và >> được đa năng hóa để nhận các mục dữ liệu của kiểu cụ thể. Nếu dữ liệu bất ngờ được xử lý, các cờ hiệu lỗi khác nhau được thiết lập mà người dùng có thể kiểm tra để xác định nếu một thao tác nhập/xuất thành công hoặc thất bại. Phần sau chúng ta sẽ khảo sát kỹ hơn.

V. NHẬP/XUẤT KHÔNG ĐỊNH DẠNG VỚI READ(),GCOUNT() VÀ WRITE()

Nhập/xuất không định dạng được thực hiện với các hàm thành viên **istream::read()** và **ostream::write()**.

Hàm **istream::read()**:

istream& read(unsigned char* *puch***, int** *nCount***);**

istream& read(signed char* *psch***, int** *nCount***);**

Trích các byte từ dòng cho đến khi giới hạn *nCount* đạt đến hoặc cho đến khi end- of-file đạt đến. Hàm này có ích cho dòng nhập nhị phân.

Hàm **ostream::write()**:

ostream& write(const unsigned char* *puch***, int** *nCount***);**

ostream& write(const signed char* *psch***, int** *nCount***);**

Chèn *nCount* byte vào từ vùng đệm (được trỏ bởi *puch* và *psch*) vào dòng. Nếu file được mở ở chế độ text, các ký tự CR có thể được chèn vào. Hàm này có ích cho dòng xuất nhị phân. Chẳng hạn:

```
char Buff[]="HAPPY BIRTHDAY";
```

```
cout.write(Buff,10);
```

Hàm **istream::gcount()**:

int gcount();

Hàm trả về số ký tự đã trích bởi hàm nhập không định dạng cuối cùng.

VI. DÒNG NHẬP/ XUẤT FILE

Để thực thi xử lý file trong C++, các chương trình phải include tập tin **<iostream.h>** và **<fstream.h>**. Header **<fstream.h>** gồm định nghĩa cho các lớp dòng **ifstream** cho nhập (đọc) từ một file, **ofstream** cho xuất (ghi) tới một file) và **fstream** cho nhập/xuất (đọc/ghi) tới một file. Các file được mở bằng cách tạo các đối tượng của các lớp dòng này. Cây phả hệ của các lớp này ở hình 8.2.

- Constructor của lớp **ofstream**:

  (1) **ofstream();**

  (2) **ofstream(const char*** *szName*,**int** *nMode*=**ios::out,int** *nProt*=**filebuf::openprot);**

  (3) **ofstream(int** *fd*);

  (4) **ofstream(filedesc** *fd*, **char*** *pch*, **int** *nLength*);

Trong đó: *szName*: Tên file được mở.

nMode: Một số nguyên chứa các bit mode định nghĩa là kiểu liệy kê của **ios**. Có thể kết hợp bằng toán tử |. Tham số này có thể một trong các giá trị sau:

Mode	Ý nghĩa
ios::app	Hàm di chuyển con trỏ file tới end-of-file. Khi các byte mới được ghi lên file, chúng luôn luôn nối thêm vào cuối, ngay cả vị trí được di chuyển với hàm **ostream::seekp()**.
ios::ate	Hàm di chuyển con trỏ file tới end-of-file. Khi byte mới đầu tiên được ghi lên file, chúng luôn luôn nối thêm vào cuối, nhưng khi các byte kế tiếp được ghi, chúng ghi vào vị trí hiện hành.
ios::in	Mở file để đọc.Với dòng **ifstream**, việc mở file đương nhiên được thực hiện ở chế độ này.
ios::out	Mở file để đọc.Với dòng **ofstream**, việc mở file đương nhiên được thực hiện ở chế độ này.
ios::trunc	Xóa file hiện có trên đĩa và tạo file mới cùng tên. Cũng có hiểu đây là chặt cụt file cũ, làm cho kích thước của nó bằng 0, chuẩn bị ghi nội dung mới. Mode này được áp dụng nếu **ios::out** được chỉ định và **ios::app**, **ios::ate**, và **ios::in** không được chỉ định.
ios::nocreate	Nếu file không tồn tại thì thao tác mở thất bại.
ios::noreplace	Nếu file tồn tại thì thao tác mở thất bại.
ios::binary	Mở file ở chế độ nhị phân (mặc định là ở chế độ văn bản).

nProt: Đặc tả chế độ bảo vệ file.

fd: Mã nhận diện file.

pch: Con trỏ trỏ tới vùng dành riêng chiều dài *nLength*. Giá trị **NULL** (hoặc *nLength*=0) dẫn đến dòng không vùng đệm.

nLength: Chiều dài tính theo byte của vùng dành riêng (0=không vùng đệm).

Dạng (1) xây dựng một đối tượng **ofstream** mà không mở file.

Dạng (2) xây dựng một đối tượng **ofstream** và mở file đã chỉ định.

Dạng (3) xây dựng một đối tượng **ofstream** và gắn (attach) với một file mở.

Dạng (4) xây dựng một đối tượng **ofstream** mà liên kết với đối tượng **filebuf**. Đối tượng **filebuf** được gắn tới file mở và vùng dành riêng.

- Constructor của lớp **ifstream**:

  (1) **ifstream();**

  (2) **ifstream(const char*** *szName*,**int** *nMode*=**ios::in,int** *nProt*=**filebuf::openprot);**

  (3) **ifstream(int** *fd*);

  (4) **ifstream(filedesc** *fd*, **char*** *pch*, **int** *nLength*);

Dạng (1) xây dựng một đối tượng **ifstream** mà không mở file.

Dạng (2) xây dựng một đối tượng **ifstream** và mở file đã chỉ định.

Dạng (3) xây dựng một đối tượng **ifstream** và gắn (attach) với một file mở.

Dạng (4) xây dựng một đối tượng **ofstream** mà liên kết với đối tượng **filebuf**. Đối tượng **filebuf** được gắn tới file mở và vùng dành riêng.

- Constructor của lớp **fstream**:

  (1) **fstream();**

  (2) **fstream(const char*** *szName*,**int** *nMode*,**int** *nProt*=**filebuf::openprot);**

  (3) **fstream(int** *fd*);

  (4) **fstream(filedesc** *fd*, **char*** *pch*, **int** *nLength*);

Dạng (1) xây dựng một đối tượng **fstream** mà không mở file.

Dạng (2) xây dựng một đối tượng **fstream** và mở file đã chỉ định.

Dạng (3) xây dựng một đối tượng **fstream** và gắn (attach) với một file mở.

Dạng (4) xây dựng một đối tượng **ofstream** mà liên kết với đối tượng **filebuf**. Đối tượng **filebuf** được gắn tới file mở và vùng dành riêng.

Nếu chúng ta sử dụng constructor ở dạng (1) thì chúng ta dùng hàm **open()** để mở file:

- Hàm **ofstream::open()**:

  void open(const char* *szName*,**int** *nMode*=**ios::out,int** *nProt*=**filebuf::openprot);**

Hàm **ifstream::open()**:

void open(const char* *szName*,**int** *nMode*=**ios::in,int** *nProt*=**filebuf::openprot);**

Hàm **fstream::open()**:

void open(const char* *szName*,**int** *nMode*,**int** *nProt*=**filebuf::openprot);**

Để đóng file chúng ta dùng hàm **close()**, hàm này ở các lớp **ifstream**, **ofstream**, và **fstream** đều có dạng:

void close();

Các hàm liên quan đến con trỏ file:

- Lớp **istream**:

Hàm **seekg()**: (*seek get*)

(1) **istream& seekg(streampos** *pos***);**

(2) **istream& seekg(streamoff** *off***,ios::seek_dir** *dir***);**

Trong đó:

+ *pos*: Vị trí mới. **streampos** là tương đương **typedef** với **long**.

+ *off*: Giá trị offset mới. là tương đương **typedef** với **long**.

+ *dir*: hướng seek. Có một trong các trị sau:

ios::begin	Seek từ bắt đầu của dòng.
ios::cur	Seek tư øvị trí hiện hành của dòng
ios::end	Seek từ cuối của dòng

Hàm **tellg()**: (*tell get*)

streampos tellg();

Hàm trả về vị trí hiện hành của con trỏ file.

- Lớp **ostream**:

Hàm **seekp()**: (*seek put*)

(1) **ostream& seekp(streampos** *pos***);**

(2) **ostream& seekp(streamoff** *off***,ios::seek_dir** *dir***);**

Hàm **tellp()**: (*tell put*)

streampos tellp();

Hàm trả về vị trí hiện hành của con trỏ file.

VI.1. Nhập/xuất file văn bản

Nếu dòng được gắn với file văn bản, việc nhập/xuất file được thực hiện một cách đơn giản bởi các toán tử >> và <<, giống như khi chúng ta làm việc với **cin** và **cout**. File văn bản chứa dữ liệu ở dạng mã ASCII, kết thúc bởi ký tự EOF.

Ví dụ 8.28: Tạo file văn bản có thể được sử dụng trong hệ thống có thể nhận được các tài khoản để giúp đỡ quản lý tiền nợ bởi các khách hàng tín dụng của công ty. Mỗi khách hàng, chương trình chứa một số tài khoản, tên và số dư (balance).

```
1: //Chương trình 8.28
2: #include <iostream.h>
3: #include <fstream.h>
4: #include <stdlib.h>
5:
6: int main()
7: {
8: ofstream OutClientFile("clients.dat", ios::out);
9: if (!OutClientFile)
10: {
```

```
11: cerr << "File could not be opened" << endl;
12: exit(1);
13: }
14: cout << "Enter the Account, Name, and Balance." << endl
15:          << "Enter EOF to end input." << endl << "? ";
16: int Account;
17: char Name[10];
18: float Balance;
19: while (cin >> Account >> Name >> Balance)
20: {
21: OutClientFile << Account << " " << Name
22:          << " " << Balance << endl;
23: cout << "? ";
24: }
25: OutClientFile.close();
26: return 0;
27: }
```

Chúng ta chạy ví dụ 8.28, kết quả ở hình 8.30

```
Enter the Account, Name, and Balance.
Enter EOF to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.26
? ^Z
```

Hình 8.30: Kết quả của ví dụ 8.28

Ví dụ 8.29: Đọc file văn bản tạo ở ví dụ 8.28 và xuất ra màn hình.

```
1: //Chương trình 8.29
2: #include <iostream.h>
3: #include <fstream.h>
4: #include <iomanip.h>
5: #include <stdlib.h>
6:
7: void OutputLine(int, char*, float);
8:
9: int main()
10: {
11: ifstream InClientFile("clients.dat", ios::in);
12: if (!InClientFile)
13: {
14: cerr << "File could not be opened" << endl;
15: exit(1);
16: }
17: int Account;
18: char Name[10];
19: float Balance;
20: cout << setiosflags(ios::left) << setw(10) << "Account"
21:      << setw(13) << "Name" << "Balance" << endl;
22: while (InClientFile >> Account >> Name >> Balance)
23: OutputLine(Account, Name, Balance);
24: InClientFile.close();
25: return 0;
```

```
26: }
27:
28: void OutputLine(int Acct, char *Name, float Bal)
29: {
30: cout << setiosflags(ios::left) << setw(10) << Acct
31:   << setw(13) << Name << setw(7) << etprecision(2)
32:<< setiosflags(ios::showpoint | ios::right) << Bal << endl;
33: }
```

Chúng ta chạy ví dụ 8.29, kết quả ở hình 8.31

```
Account  Name     Balance
100      Jones     24,98
200      Doe      345,67
300      White      0,00
400      Stone    -42,16
500      Rich     224,26
```

Hình 8.31: Kết quả của ví dụ 8.29

Do lớp **ifstream** dẫn xuất từ lớp **istream** nên chúng ta có thể dùng các hàm **istream::get()**, **istream::getline()**.

Ví dụ 8.30: Đọc file văn bản tạo ở ví dụ 8.28 bằng hàm **istream::getline()** và xuất ra màn hình.

```
1: //Chương trình 8.30
2: #include <iostream.h>
3: #include <fstream.h>
4: #include <iomanip.h>
5: #include <stdlib.h>
6:
7: #define MAXLINE 256
8:
9: int main()
10: {
11: ifstream InClientFile("clients.dat", ios::in);
12: if (!InClientFile)
13: {
14: cerr << "File could not be opened" << endl;
15: exit(1);
16: }
17: char Line[MAXLINE];
18: while (!InClientFile.eof())
19: {
20: InClientFile.getline(Line,MAXLINE-1);
21: cout<<Line<<endl;
22: }
23: InClientFile.close();
24: return 0;
25: }
```

Chúng ta chạy ví dụ 8.30, kết quả ở hình 8.32.(nội dung tập tin clients.dat)

```
100 Jones 24,98
200 Doe 345,670013
300 White 0
400 Stone -42,16
500 Rich 224,259995
```

Hình 8.32: Kết quả của ví dụ 8.30

VI.2. Nhập/xuất file nhị phân

Đối với file nhị phân (còn gọi là file truy cập ngẫu nhiên), chúng ta mở ở chế độ **ios::binary**. Đối với file nhị phân, chúng ta có thể dùng hàm **istream::get()** và **ostream::put()** để đọc và ghi từng byte một. Để đọc và ghi nhiều byte cùng lúc chúng ta có thể dùng **istream::read()** và **ostream::write()**.

Ví dụ 8.31: Lấy lại ví dụ 8.28 nhưng lưu dữ liệu dưới dạng nhị phân.

```
1: //Chương trình 8.31
2: #include <iostream.h>
3: #include <fstream.h>
4: #include <stdlib.h>
5:
6: typedef struct
7: {
8: int Account;
9: char Name[10];
10: float Balance;
11: }Client;
12:
13: int main()
14: {
15: ofstream OutClientFile("credit.dat", ios::out|ios::binary);
16: if (!OutClientFile)
17: {
18: cerr << "File could not be opened" << endl;
19: exit(1);
20: }
21: cout << "Enter the Account, Name, and Balance." << endl
22:          << "Enter EOF to end input." << endl << "? ";
23: Client C;
24: while (cin >> C.Account >> C.Name >> C.Balance)
25: {
26: OutClientFile.write((char *)&C,sizeof(Client));
27: cout << "? ";
28: }
29: OutClientFile.close();
30: return 0;
31: }
```

Chúng ta chạy ví dụ 8.31, kết quả ở hình 8.33 (nội dung tập tin credit.dat)

```
Enter the Account, Name, and Balance.
Enter EOF to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.26
? ^Z
```

Hình 8.33: Kết quả của ví dụ 8.31

Ví dụ 8.32: Đọc file tạo ở ví dụ 8.31 và xuất ra màn hình.

```
1: //Chương trình 8.32
2: #include <iostream.h>
3: #include <fstream.h>
4: #include <iomanip.h>
5: #include <stdlib.h>
6:
```

```
7: typedef struct
8: {
9: int Account;
10: char Name[10];
11: float Balance;
12: }Client;
13:
14: void OutputLine(Client);
15:
16: int main()
17: {
18: ifstream InClientFile("credit.dat", ios::in|ios::binary);
19: if (!InClientFile)
20: {
21: cerr << "File could not be opened" << endl;
22: exit(1);
23: }
24: cout << setiosflags(ios::left) << setw(10) << "Account"
25:          << setw(13) << "Name" << "Balance" << endl;
26: Client C;
27: while (InClientFile.read((char *)&C,sizeof(Client)))
28: OutputLine(C);
29: InClientFile.close();
30: return 0;
31: }
32:
33: void OutputLine(Client C)
34: {
35: cout << setiosflags(ios::left) << setw(10) << C.Account
36:          << setw(13) << C.Name << setw(7) << setprecision(2)
37:          << setiosflags(ios::showpoint | ios::right)<<
C.Balance << endl;
38: }
```

Chúng ta chạy ví dụ 8.32, kết quả ở hình 8.34

```
Account  Name      Balance
100      Jones       24.98
200      Doe        345.67
300      White        0.00
400      Stone      -42.16
500      Rich       224.26
```

Hình 8.34: Kết quả của ví dụ 8.32

BÀI TẬP

Bài 1: Cài đặt toán tử >> và << cho kiểu dữ liệu mảng để nhập và xuất.
Bài 2: Cài đặt thêm toán tử >> và << cho lớp Time trong bài 5 của chương 4.
Bài 3: Cài đặt thêm toán tử >> và << cho lớp Date trong bài 6 của chương 4.
Bài 4: Viết chương trình kiểm tra các giá trị nguyên nhập vào dạng hệ 10, hệ 8 và hệ 16.
Bài 5: Viết chương trình in bảng mã ASCII cho các ký tự có mã ASCII từ 33 đến 126. Chương trình in gồm giá trị ký tự, giá trị hệ 10, giá trị hệ 8 và giá trị hệ 16.
Bài 6: Viết chương trình in các giá trị nguyên dương luôn có dấu + ở phía trước.
Bài 7: Viết chương trình tương tự như lệnh COPY của DOS để sao chép nội dung của file .
Bài 8: Viết chương trình cho biết kích thước file.
Bài 9: Viết chương trình đếm số lượng từ có trong một file văn bản tiếng Anh.

CHƯƠNG 9

HÀM VÀ LỚP TEMPLATE

Trong phần này, chúng ta tìm hiểu về một trong các đặc tính còn lại của C++, đó là template (khuôn mẫu). Các template cho phép chúng ta để định rõ, với một đoạn mã đơn giản, một toàn bộ phạm vi của các hàm có liên quan (đa năng hóa)–gọi là các hàm template-hoặc một toàn bộ phạm vi của các lớp có liên quan-gọi là lớp template.

Chúng ta có thể viết một hàm template đơn giản cho việc sắp xếp một mảng và C++ tự động phát sinh các hàm template riêng biệt mà sẽ sắp xếp một mảng **int**, sắp xếp một mảng **float**, …

Chúng ta có thể viết một lớp template cho lớp stack và sau đó C++ tự động phát sinh các lớp template riêng biệt như lớp stack của **int**, lớp stack của **float**,…

I. Các hàm template

Các hàm đa năng hóa bình thường được sử dụng để thực hiện các thao tác tương tự trên các kiểu khác nhau của dữ liệu. Nếu các thao tác đồng nhất cho mỗi kiểu, điều này có thể thực hiện mạch lạc và thuận tiện hơn sử dụng các hàm template. Lập trình viên viết định nghĩa hàm template đơn giản. Dựa vào các kiểu tham số cung cấp trong lời gọi hàm này, trình biên dịch tự động phát sinh các hàm mã đối tượng riêng biệt để xử lý mỗi kiểu của lời gọi thích hợp. Trong C, điều này có thể được thực hiện bằng cách sử dụng các macro tạo với tiền xử lý **#define**. Tuy nhiên, các macro biểu thị khả năng đối với các hiệu ứng lề nghiêm trọng và không cho phép trình biên dịch thực hiện việc kiểm tra kiểu. Các hàm template cung cấp một giải pháp mạch lạc giống như các macro, nhưng cho phép kiểm tra kiểu đầy đủ. Chẳng hạn, chúng ta muốn viết hàm lấy trị tuyệt đối của một số, chúng ta có thể viết nhiều dạng khác nhau như sau:

```
int MyAbs(int X)
{
        return X>=0?X:-X;
}
long MyAbs(long X)
{
        return X>=0?X:-X;
}
double MyAbs(double X)
{
        return X>=0?X:-X;
}
```

Tuy nhiên với các hàm này chúng ta vẫn chưa có giải pháp tốt, mang tính tổng quát nhất như hàm có tham số kiểu int nhưng giá trị trả về là double và ngược lại.

Tất cả các hàm template định nghĩa bắt đầu với từ khóa **template** theo sau một danh sách các tham số hình thức với hàm template vây quanh trong các ngoặc nhọn (< và >); Mỗi tham số hình thức phải được đặt trước bởi từ khóa **class** như:

template <class *T***>**

hoặc **template <class** *T1***, class** *T2,…***>**

Các tham số hình thức của một định nghĩa template được sử dụng để mô tả các kiểu của các tham số cho hàm, để mô tả kiểu trả về của hàm, và để khai báo các biến bên trong hàm. Phần định nghĩa hàm theo sau và được định nghĩa giống như bất kỳ hàm nào. Chú ý từ khóa **class** sử dụng để mô tả các kiểu tham số của hàm template thực sự nghĩa là "kiểu có sẵn và kiểu người dùng định nghĩa bất kỳ".

Khi đó, hàm trị tuyệt đối ở trên viết theo hàm template:

```
template <class T>
T MyAbs(T x)
{
        return (x>=0)?x:-x;
}
```

Hàm template *MyAbs()* khai báo một tham số hình thức *T* cho kiểu của một số. *T* được tham khảo như một tham số kiểu. Khi trình biên dịch phát hiện ra một lời gọi hàm *MyAbs()* trong chương trình, kiểu của tham số thứ nhất của hàm *MyAbs()* được thay thế cho *T* thông qua định nghĩa template, và C++ tạo một hàm template đầy đủ để trả về trị tuyệt đối của một số của kiểu dữ liệu đã cho. Sau đó, hàm mới tạo được biên dịch. Chẳng hạn:

```
cout<<MyAbs(-2)<<endl;

cout<<MyAbs(3.5)<<endl;
```

Trong lần gọi thứ nhất, hàm *MyAbs()* có tham số thực là **int** nên trình biên dịch tạo ra hàm **int** *MyAbs(***int***)* theo dạng của hàm template, lần thứ hai sẽ tạo ra hàm **float** *MyAbs(***float***)*.

Mỗi tham số hình thức trong một định nghĩa hàm template phải xuất hiện trong danh sách tham số của hàm tối thiểu một lần. Tên của tham số hình thức chỉ có thể sử dụng một lần trong danh sách tham số của phần đầu template.

Ví dụ 9.1: Sử dụng hàm template để in các giá trị của một mảng có kiểu bất kỳ.

```
1: //Chương trình 9.1
2: #include <iostream.h>
3:
4: template<class T>
5: void PrintArray(T *Array, const int Count)
6: {
7: for (int I = 0; I < Count; I++)
8: cout << Array[I] << " ";
9:
10: cout << endl;
11: }
12:
13: int main()
14: {
15: const int Count1 = 5, Count2 = 7, Count3 = 6;
16: int A1[Count1] = {1, 2, 3, 4, 5};
17: float A2[Count2] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
18: char A3[Count3] = "HELLO";
19: cout << "Array A1 contains:" << endl;
20: PrintArray(A1, Count1); //Hàm template kiểu int
21: cout << "Array A2 contains:" << endl;
22: PrintArray(A2, Count2); //Hàm template kiểu float
23: cout << "Array A3 contains:" << endl;
24: PrintArray(A3, Count3); //Hàm template kiểu char
25: return 0;
26: }
```

Chúng ta chạy ví dụ 9.1, kết quả ở hình 9.1

```
Array A1 contains:
1 2 3 4 5
Array A2 contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array A3 contains:
H E L L O
```

Hình 9.1: Kết quả của ví dụ 9.1

Ví dụ 9.2: Hàm template có thể có nhiều tham số.

```
1: //Chương trình 9.2
2: #include <iostream.h>
3:
4: template<class T>
5: T Max(T a, T b)
6: {
7: return (a>b)?a:b;
8: }
9:
10: int main()
11: {
12: float A,B;
13: cout<<"Enter first number:";
14: cin>>A;
15: cout<<"Enter second number:";
16: cin>>B;
17: cout<<"Maximum:"<<Max(A,B);
18: return 0;
19: }
```

Chúng ta chạy ví dụ 9.2, kết quả ở hình 9.2

```
Enter first number:4.7
Enter second number:4.5
Maximum:4.7
```

Hình 9.2: Kết quả của ví dụ 9.2

Một hàm template có thể được đa năng hóa theo vài cách. Chúng ta có thể cung cấp các hàm template khác mà mô tả cùng tên hàm nhưng các tham số hàm khác nhau. Một hàm template cũng có thể được đa năng hóa bởi cung cấp hàm non-template với cùng tên hàm nhưng các tham số hàm khác nhau. Trình biên dịch thực hiện một xử lý so sánh để xác định hàm gọi khi một hàm được gọi. Đầu tiên trình biên dịch cố gắng tìm và sử dụng một đối sánh chính xác mà các tên hàm và các kiểu tham số đối sánh chính xác. Nếu điều này thất bại, trình biên dịch kiểm tra nếu một hàm template đã có mà có thể phát sinh một hàm template với một đối sánh chính xác của tên hàm và các kiểu tham số. Nếu một hàm template như thế được tìm thấy, trình biên dịch phát sinh và sử dụng hàm template thích hợp. Chú ý xử lý đối sánh này với các template đòi yêu các đối sánh chính xác trên tất cả kiểu tham số-không có các chuyển đổi tự động được áp dụng.

II. Các lớp template

Bên cạnh hàm template, ngôn ngữ C++ còn trang bị thêm lớp template, lớp này cũng mang đầy đủ ý tưởng của hàm template. Các lớp template được gọi là các kiểu có tham số (parameterized types) bởi vì chúng đòi hỏi một hoặc nhiều tham số để mô tả làm thế nào tùy chỉnh một lớp template chung để tạo thành một lớp template cụ thể.

Chúng ta cài đặt một lớp *Stack*, thông thường chúng ta phải định nghĩa trước một kiểu dữ liệu cho từng phần tử của stack. Nhưng điều này chỉ mang lại sự trong sáng cho một chương trình và không giải quyết được vấn đề tổng quát. Do đó chúng ta định nghĩa lớp template *Stack*.

Ví dụ 9.3:

File TSTACK.H:

```
1: //TSTACK.H
2: //Lớp template Stack
3: #ifndef TSTACK_H
4: #define TSTACK_H
5:
6: #include <iostream.h>
7:
8: template<class T>
9: class Stack
10: {
11: private:
12: int Size; //Kích thước stack
13: int Top;
14: T *StackPtr;
15: public:
16: Stack(int = 10);
17: ~Stack()
18: {
19: delete [] StackPtr;
20: }
21: int Push(const T&);
22: int Pop(T&);
23: int IsEmpty() const
24: {
25: return Top == -1;
26: }
27: int IsFull() const
28: {
29: return Top == Size - 1;
30: }
31: };
32:
33: template<class T>
34: Stack<T>::Stack(int S)
35: {
36: Size = (S > 0 && S < 1000) ? S : 10;
37: Top = -1;
38: StackPtr = new T[Size];
39: }
40:
41: template<class T>
42: int Stack<T>::Push(const T &Item)
43: {
44: if (!IsFull())
45: {
46: StackPtr[++Top] = Item;
47: return 1;
48: }
49: return 0;
50: }
51:
```

```
52: template<class T>
53: int Stack<T>::Pop(T &PopValue)
54: {
55: if (!IsEmpty())
56: {
57: PopValue = StackPtr[Top--];
58: return 1;
59: }
60: return 0;
61: }
62:
63: #endif
```

File CT9_3.CPP:

```
1: //CT9_3.CPP
2: //Chương trình 9.3
3: #include "tstack.h"
4:
5: int main()
6: {
7: Stack<float> FloatStack(5);
8: float F = 1.1;
9: cout << "Pushing elements onto FloatStack" << endl;
10: while (FloatStack.Push(F))
11: {
12: cout << F << ' ';
13: F += 1.1;
14: }
15:cout << endl << "Stack is full. Cannot push " << F << endl
16:    << endl << "Popping elements from FloatStack" << endl;
17: while (FloatStack.Pop(F))
18: cout << F << ' ';
19: cout << endl << "Stack is empty. Cannot pop" << endl;
20: Stack<int> IntStack;
21: int I = 1;
22: cout << endl << "Pushing elements onto IntStack" << endl;
23: while (IntStack.Push(I))
24: {
25: cout << I << ' ';
26: ++I ;
27: }
28:cout << endl << "Stack is full. Cannot push " << I << endl
29: << endl << "Popping elements from IntStack" << endl;
30: while (IntStack.Pop(I))
31: cout << I << ' ';
32: cout << endl << "Stack is empty. Cannot pop" << endl;
33: return 0;
34: }
```

Chúng ta chạy ví dụ 9.3, kết quả ở hình 9.3

```
Pushing elements onto FloatStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from FloatStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto IntStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from IntStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

Hình 9.3: Kết quả của ví dụ 9.3

Hàm thành viên định nghĩa bên ngoài lớp template bắt đầu với phần đầu là

template <class *T*>

Sau đó mỗi định nghĩa tương tự một định nghĩa hàm thường ngoại trừ kiểu phần tử luôn luôn được liệt kê tổng quát như tham số kiểu *T*. Chẳng hạn:

```
template<class T>
int Stack<T>::Push(const T &Item)
{
        …….
}
```

Ngôn ngữ C++ còn cho phép chúng ta tạo ra các lớp template linh động hơn bằng cách cho phép thay đổi giá trị của các thành viên dữ liệu bên trong lớp. Khi đó lớp có dạng của một hàm với tham số hình thức.

BÀI TẬP

Bài 1: Viết hàm template trả về giá trị trung bình của một mảng, các tham số hình thức của hàm này là tên mảng và kích thước mảng.

Bài 2: Cài đặt hàng đợi template.

Bài 3: Cài đặt lớp template dùng cho cây nhị phân tìm kiếm (BST).

Bài 4: Cài đặt lớp template cho vector để quản lý vector các thành phần có kiểu bất kỳ.

Bài 5: Viết hàm template để sắp xếp kiểu dữ liệu bất kỳ.

Bài 6: Trong C++, phép toán new được dùng để cấp phát bộ nhớ, khi không cấp phát được con trỏ có giá trị NULL. Hãy cài đặt lại các lớp Matrix và Vector trong đó có bổ sung thêm thành viên là lớp exception với tên gọi là Memory để kiểm tra việc cấp phát này.

TÀI LIỆU THAM KHẢO

[1] Lập trình hướng đối tượng C++ của Nguyễn Thanh Thuỷ

[2] Lập trình hướng đối tượng C++ của Trần Văn Lăng

[3] C++ Kỹ thuật và Ứng dụng – Scott Robert Ladd

[4] Ngôn ngữ lập trình C và C++

[5] Bài tập Lập trình hướng đối tượng - Nguyễn Thanh Thuỷ

[6] Introduction to Object-Oriented Programming Using C++ - **Peter Müller**

[7] ..